An XML-based Quality of Service Enabling Language for the Web

Xiaohui Gu, Klara Nahrstedt, Wanghong Yuan, Duangdao Wichadakul
Department of Computer Science
University of Illinois at Urbana-Champaign
xgu, klara, wyuan1, wichadak@cs.uiuc.edu

Dongyan Xu
Department of Computer Sciences
Purdue University
dxu@cs.purdue.edu

Abstract

In this paper, we introduce an XML-based Hierarchical QoS Markup Language, called HQML, to enhance distributed multimedia applications on the World Wide Web (WWW) with Quality of Service (QoS) capability. The design of *HQML* is based on two observations: (1) the absence of a systematic QoS specification language, that can be used by distributed multimedia applications on the WWW to utilize the state-of-the-art QoS management technology; and (2) the power and popularity of XML to deliver richly structured contents over the Web. HQML allows distributed multimedia applications to specify all kinds of application-specific QoS policies and requirements. During runtime, the *HQML Executor* translates the HQML file into desired data structures and cooperates with the *QoS proxies* that assist applications in end-to-end QoS negotiation, setup and enforcement. In order to make QoS services tailored toward user preferences and meet the challenges of uncertainty in the distributed heterogeneous environments, the design of HQML is featured as *interactive* and *flexible*. In order to allow application developers to create HQML specifications correctly and easily, we have designed and developed a unified visual QoS programming environment, called *QoSTalk*. In *QoSTalk*, we adopt a grammatical approach to perform consistency check on the visual QoS specifications and generate HQML files automatically. Finally, we introduce the *distributed QoS compiler*, which performs the automatic mappings between application and resource level QoS parameters to relieve the application developer of the burden of dealing with low level QoS specifications.

keywords

Quality of Service, XML, Distributed Multimedia Applications, Visual Programming Environment

1 Introduction

Future computing systems have been envisioned as ubiquitous, pervasive and nomadic [42, 3, 25]. They will consist of devices that are diverse in size, capability, and power consumption. People view videos or join video

conferencing on the Web using laptops, Personal Digital Assistants (PDAs) or even Cell-Phones. The emergence of wireless network further increases the heterogeneity of current computing environment. Nomadic users hope to receive consistent, predictable, timely, and reliable services on the World Wide Web (WWW) in spite of the fluctuation and shortage of underlying resources (e.g., network congestion). In order to realize such a vision, Web multimedia applications need specification and provision of Quality of Service (QoS). By "QoS", we mean not only the specification and provision of proper *synchronization* and *integration* of multimedia streams, which are extensively addressed by different multimedia languages [7, 10], but also the specification and provision of **predictability** (the ability to maintain the contracted QoS and have minimum probability of QoS violations during the resource fluctuation period), **continuity** (the ability to degrade gracefully and adjust resource allocation distributions dynamically to tolerate transient resource scarcity) and **accessibility** (the ability to access the service from a wide range of devices, including PCs, workstations, cell-phones, and PDAs).

A wealth of research work has been done to support Quality of Service for distributed multimedia applications. Researchers provided solutions for setup and enforcement of QoS in the networks, in the operating system (OS), in applications themselves, and most recently in the middleware systems which reside between applications and OS. The network and OS solutions definitely help, but they may not be easily and rapidly deployed, hence they may not keep up with the explosive growth of the Internet. The application-level solutions, such as adaptive source coding [38] and tightly-coupled application control of critical QoS parameters [8], yield (1) complex implementation of applications, and (2) difficult adjustment or full re-implementation of applications to new devices, and their underlying operating systems and networks. In recent years, middleware solutions have evolved to overcome the above shortcomings. QoS middleware systems assist multimedia applications in QoS setup and enforcement by utilizing QoS services in networks and operating systems if available, or by providing adaptation services if best effort services exist only. Two major types of QoS middleware systems have been developed: (1) *Reservation-based Systems* [27, 34] get the QoS specifications in the form of system resource requirements, reserve the specified resources and enforce the delivery of requested QoS during runtime; and (2) *Adaptation-based Systems* [6, 4, 29] get the QoS specifications in the form of bounds on resource utilization, application-specific adaptation rules, and adapt resource allocations according to changes in resource availability. Recently developed reconfigurable component-based QoS middleware systems [26, 35, 22, 41] combine these two mechanisms together. In these systems, QoS management mechanisms (e.g., reservation, adaptation, reconfiguration) and QoS policies, are separated in order to reduce the burden on application developers and enable generic QoS middleware systems. In this paper, we call those generic QoS middleware entities the *QoS Proxies*. They should provide QoS management services (e.g., negotiation, adaptation, (re)configuration, resource allocation and reservation, service/host discovery, etc.) for the application according to the application-specific QoS specifications.

Although so many QoS solutions are available, application developers still cannot create QoS-aware multimedia

applications easily, especially for the Web applications. The reason lies in the fact that a universal QoS specification language is still absent. Although different QoS specification languages have been developed, they are either tightly coupled with a specific programming language or cannot be extended easily to catch up with the rapid development of new QoS services. On the other hand, the traditional Web languages like HTML cannot be used and extended to fulfill the task of QoS specification. Although new multimedia languages (e.g., TAOML [7], SMIL [10]) have been proposed to address the increasing requirements of distributed multimedia applications on the WWW, their solutions are still limited to the synchronization and integration issues and do not provide interactions with generic QoS management systems (QoS Proxies) to address the quality guarantees and adaptation issues in case of resource fluctuation and scarcity in the distributed heterogeneous environments. The Extensible Markup Language (XML) [9] is an ideal QoS specification language, that can be used by distributed Web multimedia applications, because it is the universal format for structured documents and data on the Web and also extensible. In addition, we can use XML query language [12] to access and lookup the XML-based QoS specification on the WWW very easily. However, XML itself does not tell application developers how to specify QoS requirements for his/her applications. We must define a minimum set of suitable tags to allow application developers to express their QoS requirements and policies, based on the XML syntax. Application developers are also allowed to define their own service-specific tags.

In this paper, we introduce an XML-based QoS enabling language for the WWW, called HQML, an acronym for "Hierarchical QoS Markup Language". The HQML specifications are classified into three levels.

- **User Level** HQML specifications provide tags to specify qualitative QoS criteria (e.g., high, low, average), user focus of attention (e.g., smoothness, clarity), prices for the required services and the price model (e.g., flat rate, per transmitted byte charges) the service provider adopts. The user level QoS specifications are used, during runtime, to find the best "match" between the user's economic condition, preferred QoS level and the available QoS levels provided by different service providers. We do not expect users to give very detailed quantitative specifications of all kinds of application-specific QoS parameters which may potentially be very complex.

- **Application Level** HQML specifications provide tags to specify all kinds of application level QoS parameters (e.g., frame rate, frame size, resolution, etc.), application-specific QoS policies (e.g., adaptation rules, reconfiguration rules). For distributed Web multimedia applications, HQML also provides tags to describe their application configurations, which are a set of application components connected into directed acyclic graphs. The QoS specifications of this level are used by the QoS Proxies (e.g., adaptor, configurator) to set up and enforce the QoS on behalf of the application even if the underlying OS and network QoS support is absent.

- **System Resource Level** HQML specifications provide tags to specify different system resource requirements. (e.g., memory, cpu, network bandwidth, power, etc.). If OS and network QoS management services are available, the QoS Proxies could initiate the reservation of the required end-to-end resources on behalf of the application

according to this level's QoS specifications.

In order to improve QoS provisions automatically, based on history data and user's preferences, HQML provides special tags to enable the *interactions* between the user and QoS Proxies. Application developers could use those tags to specify under what circumstances a particular notification should be sent to the user (e.g., if a certain adaptation or reconfiguration happens) or a specific feedback is desired from the user. (e.g., "satisfaction", "dissatisfaction"). These feedbacks are used to derive users preference profiles and improve the satisfaction degree of QoS provisions (e.g., optimization of adaptation rules) based on AI methods for learning rules (e.g., Neural Networks) [20]. Furthermore, the syntax of HQML is designed as flexible as possible to enable the highest accessibility of QoS-aware multimedia services on the WWW. For example, there may be services available to an application at run-time that are not known or available to the application developer at design-time, but may be useful for multimedia applications. Thus, the application developer should be allowed to abstractly specify *optional* services that, if present at runtime, enhance the application. But if the optional services are not available, the application should be allowed to start as well.

Although HQML follows standard XML syntax and can be used very easily, several critical issues require careful considerations. First, some information in HQML specifications cannot be derived directly. For example, the application developer may not know the system resource requirements for his/her applications in advance. For the adaptation rules, the application developer needs to specify the threshold values of each adaptation triggers, which will actually decide the activation timing of each adaptation choice. But those threshold values may not be easily derived. Second, we need to check the consistency or accuracy of HQML specifications. Since application developers are allowed to use HQML to specify their own QoS requirements and policies, any illegal specifications may break down the underlying systems. Although document type definition (DTD) [9] can be used to check some errors in the XML-based files, it is far from enough for the QoS specifications. For example, we must make sure that there is no deadlock or starvation in the specifications of application configurations for a distributed multimedia application. Moreover, the QoS parameters between two connected components must be consistent. For example, if an MPEGII encoder is connected with an H261 decoder, or a low quality video player is connected with a high performance video server, then the application will not work properly. We address those problems by introducing a visual QoS programming environment, called *QoSTalk*, which assists application developers to generate HQML files correctly and easily.

Finally, we introduce the HQML Executor module that is responsible for translating the HQML specifications into desired data structures and cooperating with QoS proxies to provide QoS for the Web multimedia applications. The HQML Executor can be installed into any user-preferred Web browser in advance. Our approach does not require any major re-implementation of the legacy Web multimedia applications. Application developers are relieved from the burden of implementing QoS "knowledge" in their applications themselves. Instead, they use HQML to specify their application-specific QoS policies and requirements and delegate the responsibility of QoS provisions to the QoS

Proxies. By following this approach, the QoS can be provided in a more fair and efficient way because the QoS Proxies have the global knowledge about the system resource conditions and control multiple applications simultaneously. Our approach does not assume any specific QoS middleware framework and can be applied to any of them as long as they provide generic QoS middleware services (negotiation, adaptation, configuration, monitoring, resource reservation (optional)). Since HQML is based on the XML syntax, new tags can be incorporated very easily to utilize any emerging QoS services via self-describing, extensible nature of XML.

The rest of the paper is organized as follows. In section 2, we present the design of HQML in detail. In section 3, we present the Visual QoS Programming Environment *QoSTalk*. In section 4, we introduce the *HQML Executor*. In section 5, we present the initial experimental results from the *HQML Executor* and *QoSTalk* prototype. In section 6, we review related works about multimedia languages and QoS specifications . Section 7 concludes this paper.

2 HQML: XML-based Hierarchical QoS Markup Language

2.1 Application Model

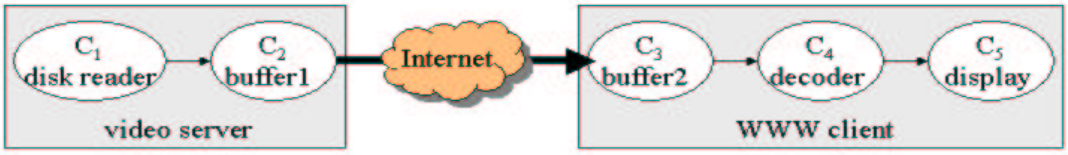


Figure 1: Application Configuration for Video-On-Demand Application.

We first introduce the application model upon which the design of HQML is based. We consider a generic *application component model* to characterize the structure of distributed multimedia applications. All application components are constructed as *tasks*, which perform specific operations on the multimedia data passing through them, such as transformation, aggregation, prefetching and filtering. Each component accepts input with a QoS level Q^{in} and generates output with a QoS level Q^{out}, both of which are vectors of application-level QoS parameter values. In order to process input and generate out, a specific amount of resources is required. The multimedia data can be either basic multimedia objects, such as text, image, video/audio streams or composite objects containing multiple media types. Tasks can be connected into a directed acyclic graph (DAG), which is called an *application configuration*. The application configuration is the "flow chart" upon which multimedia data flows between the service providers and the end user. Finally, all multimedia objects, received by the end host, are synchronized (if necessary) and presented to the Web user by using any existing multimedia languages or softwares (e.g., SMIL, TAOML, Authoring Systems). For example, in the video-on-demand application illustrated in Figure 1, data is read from the disk, stored in buffers at the sender side, transmitted over the network and again stored in the receiver buffer, and then decoded before presenting

to the Web user. The buffers are regarded as application components rather than memory storage so that we can utilize different buffer management schemes explicitly.

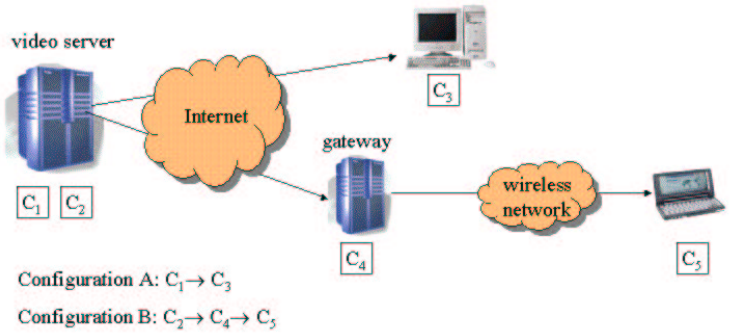


Figure 2: Service Polymorphism in Video-On-Demand Application.

The design of HQML is based on the important QoS concept of "Service Polymorphism". More precisely, the same distributed multimedia Web services can be delivered in multiple forms and formats, using different application configurations from the service provider to different types of clients, and possibly through an intermediate gateway (infrastructure proxy) [17, 45]. Different application configurations provide various quality levels or similar quality levels but have distinct resource requirements. For example, a simple *Video On Demand* application on the WWW can have two different configurations. For a powerful desktop client, connected to a high-speed LAN, the application configuration may only contain two components, a MPEG Video Server (service provider) and a MPEG Video Player including the MPEG decoder (WWW user). For a resource-constrained device, like Handheld PC connected with a wireless network, however, the application configuration may be changed to include three components, a MPEG Video Server (service provider), a MPEG to Bitmap Transcoder (gateway), and a Bitmap Player (WWW user). The second configuration delivers lower or similar quality, depending on the performance of transcoder, but requires less computation resource on the WWW client. Figure 2 illustrates the above concept. For other complex distributed multimedia applications like *Video Conferencing*, more diversified application configurations may exist to accommodate the large range of client capabilities.

We adopt a hierarchical approach to characterize the application configurations. The hierarchical approach helps accommodate the scalability problem for the complex distributed multimedia applications. [21] The basic building block of the configuration is called *atomic component*, which only contains one basic multimedia function, (e.g., *MPEGII Decoder*, *MPEG to Bitmap Transcoder*, *Prefetcher*, etc.). A collection of interconnected *atomic components* forms a *service* on a single host, which is called *compound component*. Beyond a single end host, we group the entire distributed application into *clients*, *gateways*, *servers* and *peers*, with each of them running on one of networked hosts. A group of interconnected hosts of the same type, such as *servers or gateways*, forms a *cluster*. The connections between these components, which are called *links*, represent the media transfer flows. We have designed three different

link types: (1) Fixed Links; (2) Mobile Host Links; and (3) Mobile User Links. A fixed link defines a wired data communication channel, which cannot be interrupted or "moved" during the runtime. A mobile host link defines a wireless communication channel, which means the end host could move within certain range during runtime. A mobile user link is defined to specify the user mobility, which means the user could move from one machine to another during the runtime. When the user moves from the old machine to a new one during the runtime of an application, such as the *Video On Demand* application, 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, from the interruption point, automatically. All of these links could be one-way or two-way connections. The media data flow between two atomic components in a single host is also defined as the fixed link.

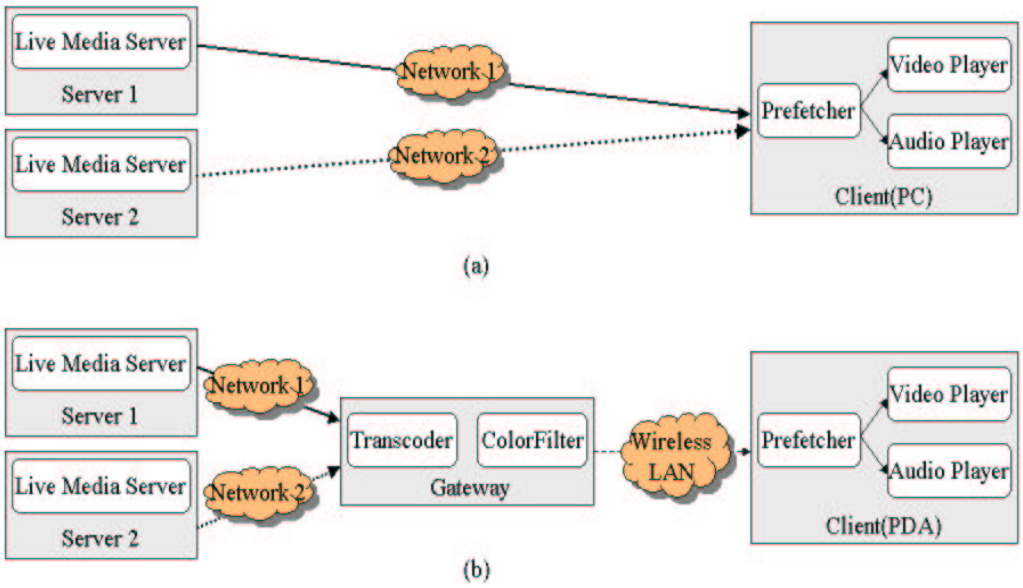


Figure 3: Two Hierarchical Configurations for the Live Media Streaming Application.

Figure 3 (a) and (b) illustrate two possible hierarchical configurations for the *Live Media Streaming* application. The configuration (a) delivers higher quality (for higher price) to a client with high resource availability, such as a powerful PC connected to the MBone. The "server2" is a mirror site of the primary live media server "server1". The QoS proxies could automatically switch from the server 1 to server 2 when the primary server is overloaded. The dotted line represents an alternative route. The configuration (b) delivers lower yet acceptable quality (for lower price) to a client with low resource availability, such as a PDA connected with a wireless network. An intermediate gateway performs transformations and necessary degradations to meet capabilities of resource-constrained devices like PDA. We will use this *Live Media Streaming* application as an example throughout the paper.

2.2 XML Overview

We now provide a short overview of XML. It is a markup language for documents containing structured information. Structured information contains both content (words, pictures, etc.) and some indication of what role that content plays (for example, content in a section heading has a different meaning from content in a footnote.). Almost all documents have some structure. A markup language is a mechanism to identify structures in a document. The XML specification defines a standard way to add markup to documents. Unlike HTML, the set of tags in XML is flexible; the tag syntax is defined by a document's associated DTDs. In fact, XML is really a meta-language for describing markup languages. In other words, XML provides a facility to define tags and the structural relationships between them. Since there's no predefined tag set, there can't be any preconceived semantics. All of the semantics of an XML document will be defined by the applications that process them.

We have chosen to build atop the XML for the HQML schema design, leveraging its allowances for the creation of customizable, application-specific markup languages. We believe there is a natural synergy between XML's need for new schema to become successful and the QoS specification requirements of Web multimedia applications. We will introduce the HQML syntax in the next section. The design of HQML schema is based on a hierarchical approach and organized into three different levels: (1) *User level*, (2) *Application level* and (3) *System resource level*. Figure 4 illustrates the three-level hierarchical structure of the HQML schema.

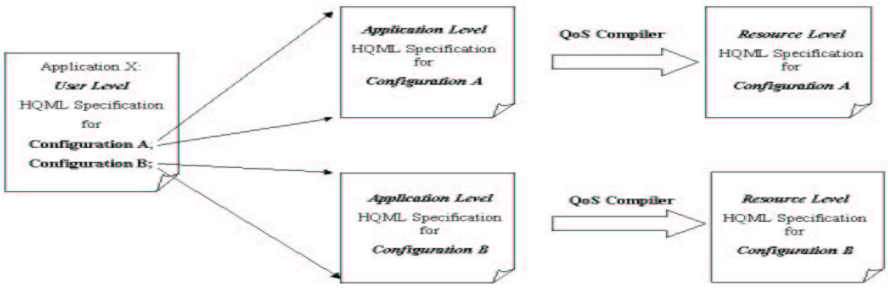


Figure 4: Three-level Hierarchical Structure of the HQML Schema.

2.3 HQML Syntax for User Level QoS Specifications

The user level QoS specifications mainly include three parts: (1) The overall descriptions about the application. (e.g., name, service provider); (2) The multiple application configurations with associated qualitative user level QoS criteria (e.g., low, average, high, smoothness, clarity) and the initial prices ($1, $5, $10); and (3) The price models which the service provider would like to use (e.g., flat rate, per transmitted byte charges, per minutes charges).

```
<App name = "Live Media Streaming" ServiceProvider = "Company X">
   <Configuration id = "100">
       <UserLevelQoS> high </UserLevelQoS>
       <QoSPreference> smoothness </QoSPreference>
       <Price unit = "$"> 5 </Price>
       <PriceModel> flat rate </PriceModel>
   </Configuration>
   <Configuration id = "101">
       <UserLevelQoS> Average </UerLevelQoS>
       <QoSPreference> smoothness </QoSPreference>
       <Price unit = "$"> 1 </Price>
       <PriceModel> flat rate </PriceModel>
   </Configuration>
   <Configuration id = "200">
       <UserLevelQoS> high </UserLevelQoS>
       <QoSPreference> clarity </QoSPreference>
       <Price unit = "$"> 2 </Price>
       <PriceModel> per hour increase <PriceModel>
   </Configuration>
</App>
```

Figure 5: Example of User Level HQML Specifications for the Live Media Streaming Application Shown in Figure 3.

The <App> tag is a container tag. It has one required attribute, “name”, which is either a string or a reference identifying the type/class of the application being described. It has several optional attributes such as “ServiceProvider”, which is a string specifying the company name of the service provider. It contains at least one <Configuration> tag, which is also a container tag. The <Configuration> tag has one required attribute, “id”, which is the identification number used to retrieve the HQML file about the corresponding application configuration. It includes one <UserLevelQoS> tag providing the qualitative description of the application configuration from a user’s point of view. It contains one <QoSPreference> tag indicating the user’s quality preference while using the multimedia services on the WWW. For example, some users may think ”smoothness” is the most important satisfaction criteria for the Video On Demand application for the purpose of entertainment. However, if the Video On Demand service is used for the remote medical surgery, the doctor may think “clarity” is the most important criteria. In the latter case, we can drop some frames (especially B or P frames of the MPEG video) to preserve the resolution of the video when the network bandwidth becomes deficient. Each <Configuration> tag also includes one <Price> tag, indicating the initial charging price, when the user invokes the multimedia service with a specific application configuration on the WWW. It also contains one <PriceModel> tag indicating the price model the service provider would like to adopt. Figure 5 gives an example of user level HQML specifications for the *Live Media Streaming* application illustrated in Figure 3. The Configuration “100” represents the application configuration of Figure 3(a). The configuration “101” is shown in Figure 3 (b).

2.4 HQML Syntax for Application Level QoS Specifications

HQML provides many tags for application level QoS specifications, which include all application level QoS parameters and policies about a particular application configuration. The specifications begin with the <AppConfig>

```
<AppConfig id = "100">
  <CriticalQoS type = "frame rate">
    <Range unit = "fps">
      <UpperBound> 40 </UpperBound>
      <LowerBound> 30 </LowerBound>
    </Range>
  </CriticalQoS>
  <ServerCluster>
    <Server type = "replacable">
      <HostAddr type = "primary">
         paris.cs.uiuc.edu
      </HostAddr>
      <HostAddr type = "alternative">
         boston.cs.uiuc.edu
      </HostAddr>
      ...
    </Server>
  </ServerCluster>
  <ClientCluster>
    <Client type = "required">
      <Hardware> Pentium PC 500 </Hardware>
      <Software> Windows 2000 </Software>
      ...
    </Client>
  </ClientCluster>
  <LinkList>
    <Link type = "FixedLink">
      <Start> Server </Start>
      <End> Client </End>
      ...
    </Link>
  </LinkList>
  <ReconfigRuleList>
    <ReconfigRule>
       <Condition type = "Bandwidth">
         very low
       </Condition>
       <ReconfigAction type = "switch to" >
           101
       </ReconfigAction>
   <ReconfigRule>
   <Notification> Reconfigured </Notification>
   <Feedback> early or late </Feedback>
 </ReconfigRuleList>
</AppConfig>
```

(a)Example Specification of Application Configuration Shown in Figure 3 (a).

```
<AppConfig id = "101">
  <CriticalQoS type = "frame rate">
    <Range unit = "fps">
      <LowerBound> 10 </LowerBound>
      <UpperBound> 20 </UpperBound>
    </Range>
  </CriticalQoS>
  <ServerCluster>
    <Server type = "replacable">
      <HostAddr type = "Primary">
          paris.cs.uiuc.edu
      </HostAddr>
      <HostAddr type = "alternative">
          boston.cs.uiuc.edu
      </HostAddr>
      ...
    </Server>
  </ServerCluster>
  <GatewayCluster>
    <Gateway type = "replacable">
      ...
    <Gateway>
  </GatewayCluster>
  <ClientCluster>
    <Client type = "required">
      ...
    <Client>
  </ClientCluster>
  <LinkList>
    <Link type = "FixedLink">
      <Start> Server </Start>
      <End> gateway </End>
      ...
    </Link>
    <Link type = "MobileHostLink">
       <Start> gateway </Start>
       <End> Client </End>
       ...
    </Link>
  </LinkList>
</AppConfig>
```

(b)Example Specification of Application Configuration Shown in Figure 3 (b).

Figure 6: Example of Application Level HQML Specifications for the Two Configurations of the Live Media Streaming Application Shown in Figure 3.

tag, which is a container tag with one required attribute, “id”, the same as that of the <Configuration> tag in the user level specifications. The <AppConfig> tag contains one <CriticalQoS> tag, zero or one <ServerCluster> tag, <GatewayCluster> tag, <ClientCluster> tag, and <PeerCluster> tag. The “server”, “gateway” and “client” are used to specify the dedicated services. We can use “peer” to describe any other generic services. The network connections among application components are specified by the <LinkList> tag. <AppConfig> could also contain one <ReconfigRuleList> tag to specify the policies for the dynamic QoS reconfiguration services. Figure 6 gives the examples of application level QoS specifications in HQML for the configurations “100” and “101” of the *Live Media Streaming* application.

The <CriticalQoS> tag specifies the *critical QoS parameter* [29] which is the most important end-to-end application-level QoS parameter protected by degrading other QoS parameters during the resource deficiency period. The critical QoS parameters are usually mapped to the user’s quality preference. This tag has one required attribute ”type” specifying the name of the parameter (e.g., frame rate, resolution). It has one internal tag <Range> specifying the allowed fluctuation range of the critical QoS parameter. The <Range> tag has one attribute ”unit” indicating the

```
<Gateway type = "replacable">
   <Hardware> Pentium 500 </Hardware>
   <Software> Windows 2000 </Software>
   <Atomic type = "Replacable">
      <Name> Transcoder </Name>
      <Method name = "getStates"/>
      <Method name = "setStates">
         <param lextype = "strings"> states </param>
      </Method>
      <Method name = "scale">
         <param lextype = "real:range = 0-1">
            scalingfactor
         </param>
      </Method>
      <InputQoSList>
         <MediaObjectList>
            <MediaObject format = "MPEGII">
               Video
          </MediaObject>
            <MediaObject format = "wav">
              Audio
          </MediaObject>
         </MediaObjectList>
         <Delay unit = "ms"> 100 </Delay>
         <Jitter unit = "ms"> 30 </Jitter>
         <LossRate unit = "%"> 5 <LossRate>
      <InputQoSList>
      <OutputQoSList>
         <MediaObjectList>
             <MediaObject format = "Bitmap">
                Video
            </Mediaobject>
            ...
         <MediaObjectList>
          ...
      </OutputQoSList>
   </Atomic>
   <Atomic type = "optional">
      <Name> ColorFilter </Name>
      <Method name = "ChangeColorDepth">
         <param lextype = "int:range = 2-256">
      </Method>
      ...
   </Atomic>
   <LinkList>
      <Link type = "FixedLink">
         <Start> Transcoder </Start>
         <End> ColorFilter </End>
      </Link>
   </LinkList>
</Gateway>
```

(a)Example of Application Level Specification for Compound Component "Gateway" Shown in Figure 3 (b).

```
<Client type = "required">
   <Hardware> PDA </Hardware>
   <Software> Windows CE </Software>
   <Atomic type = "Optinal">
      <name> Prefetcher </Name>
      <InputQoSList>
         <MediaObjectList>
            <MediaObject format = "Bitmap">
               Video
            </MediaOjbect>
            ...
         </MediaObjectList>
         <FrameRate unit = "fps">
            <UpperBound> 30 </UpperBound>
            <LowerBound> 5  </LowerBound>
         </FrameRate>

     </InputQoSList>
     ...
   </Atomic>
   <Atomic type = "Required">
      <Name> Video Player </Name>
      ...
   </Atomic>
   <Atomic type = "Required">
      <Name> Audio Player </Name>
     ...
   </Atomic>
   <LinkList>
     <Link type = "FixedLink'>
        <Start> Prefetcher </Start>
        <End> Video Player </End>
     </Link>
     ...
   <LinkList>
   <AdaptationRuleList>
     <AdaptationRule>
        <Condition type = "Bandwidth">
           low
        </Condition>
        <Action>
           <Component> ColorFilter </Component>
           <Method> ChangeColorDepth </Method>
        </Action>
        <Notification> color degrade! </Notification>
        <Feedback> early or late </Feedback>
     </AdaptationRule>
   </AdaptationRuleList>
</Client>
```

(b)Example of Application Level Specification for Compound Component "Client" Shown in Figure 3 (b).

Figure 7: Example of Application Level HQML Specifications for the Two Compound Components in Configuration “101” - “Gateway” and “Client”.

measured unit (e.g., ”ms”, ”fps (frames per second)”). The <Range> tag includes two internal tags, <UpperBound> and <LowerBound>.

The <ServerCluster>, <GatewayCluster>, <ClientCluster> and <PeerCluster> tags are all container tags and have similar internal structures. We use <GatewayCluster> as an example to explain their internal structures. Each <GatewayCluster> tag should include at least one <Gateway> tag which represents one compound application component or one host machine. The Figure 7 (a) illustrates the HQML specifications for the compound component, “Gateway” of the configuration “101”. As we mentioned in Section 1, HQML allows the application developer to abstractly specify optional services that, if present at runtime, enhance the application. Thus, each component has an attribute “type” to specify whether it is “required”, “replacable” or “optional”.

If the component has the type “required”, the QoS Proxies must discover and instantiate the component. If the component is specified as “replacable” and a QoS violation is detected, the QoS Proxies could select one or more alternate components and perform a transparent transition from the primary components to the alternative ones to maintain the initially agreed QoS. For example, when the initial video server becomes overloaded, the QoS Proxies could select one of its mirror servers to recover automatically from the QoS violations. Finally the type “optional” gives the highest flexibility to the QoS Proxies which could discover a similar service to replace it or simply neglect

it to accommodate unexpected runtime environments. The <Server>, <Gateway> and <Peer> tags can include any number of <HostAddr> tags which indicate the host addresses where all their atomic components will be dynamically downloaded and instantiated. If multiple <HostAddr> tags are included, one of them is the primary server or gateway while others are their mirror sites. Figure 6 (a) gives the example of <HostAddr> tag usage for the "Server" compound component. If there is no <HostAddr> tag included, the QoS discovery proxy will discover a suitable machine in the distributed environment to instantiate the compound component. In this case, the <Server> tag will include two internal tags, <Hardware> and <Software>, which specifies the hardware (e.g., PC, PDA) and software (e.g., Windows 2000, Solaris 5.3) requirements of executing the specified services. Figure 7 (a) gives such an example for the "Gateway" component. Thus the QoS discovery proxy will try to find a machine to instantiate the gateway compound component, which is at least a Pentium 500 PC and has installed windows 2000.

Each <Gateway> tag contains at least one <Atomic> tag, which represents an atomic application component. The <Atomic> tag also has one attribute "type" specifying whether it is "required", "replacable" or "optional". It has one internal tag <Name> to give the service name of the atomic component (e.g., "Transcoder", "ColorFilter"). Each <Atomic> tag could include any number of <Method> tags specifying the method calls that can be invoked on it. There are two default methods for each atomic component, "start" and "stop". In Figure 7 (a), three additional methods "getstates", "setstates" and "scale" are included in the atomic component "transcoder". Each <Method> tag may include several internal tags "<Param>", which describe the input parameters required by the method. For example, the "scale" method requires one input parameter "scalingfactor", which controls the resolution of the output bitmap images from the transcoder. For each atomic component, we also need to specify its input and output application level QoS parameters by using tags <InputQoSList> and <OutputQoSList>. The application developer could use <MediaObjectList> to characterize the features of the input or output media streams. The specifications may include the "media type"(e.g., text, image, audio, video), "media format" (e.g., JPEG, MPEG, Bitmap, wav) and also the temporal and spatial relationships among multiple streams. We could use TAOML [7] to describe the complex media streams. Next, we need to specify the input QoS parameters required by the atomic component and the output QoS parameters guaranteed by it. HQML provides tags like <Delay>, <Jitter>, <LossRate>, <Throughput> to fulfill the task. Application developers could also define their own application-specific QoS parameters in HQML. Then application developers could use <LinkList> tag to specify the connections between different atomic components.

Finally, each <Server>, <Gateway>, <Client> or <Peer> tag could contain zero or one <AdaptationRuleList> tag which specifies the adaptation policies the compound component will follow. In Figure 7 (b), <AdaptationRuleList> tag is used to specify the adaptation policies the client compound component of confiugration "101" follows. The <AdaptationRuleList> tag includes at least one <AdaptationRule> tag which consists of two required internal tags, <ConditionList> and <Action> tags. The <ConditionList> tag specifies a list of linguistic values (e.g., high, low, very low) for each system resources (e.g., cpu, network, power) that decide the activation timing of a certain adaptation

action. These linguistic values will be mapped into a set of threshold values according to the system resource level QoS specifications introduced in the next section. The <Action> tag includes two internal tags, <Component> and <Method> that defines which method belonging to which component will be invoked for the action. The component may reside in the local host or a remote site. In Figure 7 (b), for example, one adaptation action for the client to take, when the bandwidth drops below a certain threshold, is to ask the "ColorFilter" component in the intermediate gateway to decrease color depth.

As we mentioned in Section 1, the design of HQML is featured as *interactive*. HQML provides two tags <Notification> and <Feedback> to enable the interactions between the user and the QoS Proxies. In Figure 7 (b), for instance, the application developer specifies that a notification message "Color degrade!" should be sent to the user when that particular adaptation happens. The developer also tells the QoS Proxy to request a feedback about the timing of the adaptation from the user. These feedbacks are useful for the QoS Proxies to derive a user profile so that the QoS provisions can be tuned toward user preferences and improved with experience.

After we finish the QoS specifications for each cluster component, we need to specify how these compound components are connected into an application configuration. HQML provides the <LinkList> tag for this purpose. It includes a set of <Link> tags. Each <Link> tag has one attribute "type". As we mentioned in section 3.1, there are three different link types: (1) Fixed Link; (2) Mobile Host Link; and (3) Mobile User Link. The QoS proxy will treat them differently, such as different calculation function of end-to-end bandwidth for the wired and wireless networks. For the mobile user link, the QoS Proxy also needs to insert the persistent state manager to store the frame number, for example. Thus the video streaming session could be restarted from the interrupting point when the user moves to a new machine. The <Link> tag includes two internal tags, <Start> and <End> indicating the two end points of the connection.

The <ReconfigRuleList> is the last important internal tag contained in the <AppConfig> tag. It includes a list of <ReconfigRule> tags, which tell the QoS proxy how to dynamically reconfigure the application when the system resources drop below certain minimum bounds and cannot be hidden by the data adaptations like decreasing colordepth, dropping frames. The syntax of the <ReconfigRule> tags is similar to that of the <AdaptationRule> tags. It also consists of two internal tags, <Condition> and <Action>. However, the action here is switching to another application configuration instead of invoking some method of a component. In Figure 6 (a), the application developer specifies that when the end-to-end bandwidth between the server and the client drops to a very low degree, the QoS proxies should reconfigure the application from configuration "100" to "101" to guarantee the continuity of services. Similar to the adaptation rule specifications, the application developer could use <Notification> and <Feedback> tags here to enable the interactions between the user and the QoS Proxies. The dynamic reconfiguration usually takes relatively larger overhead and causes QoS violations. Thus it should be avoided as much as possible. Thus, the advantage of HQML is to allow the QoS proxy to always choose the optimal configuration at the beginning

rather than a fixed configuration like current distributed multimedia applications on the WWW.

2.5 HQML Syntax for Resource Level QoS Specifications

```
<AppConfig id = "100">
  ...
  <ServerCluster>
    ...
  </ServerCluster>
  <ClientCluster>
    ...
  </ClientCluster>
  <LinkList>
    <Link type = "FixedLink">
       ...
        <Throughput>
          <Average unit = "MB"> 50 </Average>
          <Burstiness unit = "MB"> 5 </Burstiness>
        </Throughput>
       <Delay unit = "ms"> 100 </Delay>
       <LossRate unit = "%"> 3 </LossRate>
       <Jitter unit = "ms"> 10 </Jitter>
       <Level> hard </Level>
    </Link>
    <Link type = "FixedLink">
    <ReconfigRuleList>
       ...
    </ReconfigRuleList>
    <ThresholdList>
       <very high type = "Bandwidth">
          <LowerBound unit = "MB">
             70
          </LowerBound>
        </very high>
       <high type = "Bandwidth">
          <LowerBound unit = "MB">
             50
          </LowerBound>
       ...
       <very low type = "Bandwidth">
          <UpperBound unit = "MB">
             5
          </UpperBound>
       </very low>
       ...
    </ThresholdList>
 </AppConfig>
```

(a) Example of Resource Level Specification for the Application Configuration Shown in Figure 3 (a).

```
<Client type = "required">
    <Hardware> PDA </Hardware>
    <Software> Windows CE </Software>
    ...
    <Atomic type = "optional">
       <Name> Prefetcher </Name>
       ...
       <CPU unit = "%">
          <Average> 30 </Average>
          <Deviation> 10 </Deviation>
       </CPU>
       <Memory unit = "KB">
           <Average> 3 </Average>
           <Deviation> 1 </Deviation>
       </Memory>
       <Disk unit = "MB">
           <Average> 16 </Average>
           <Deviation> 0 </Deviation>
       </Disk>
    </Atomic>
    ...
    <ThresholdList>
       <high type = "CPU">
          <LowerBound unit = "%">
             40
          </LowerBound>
          <UpperBound unit = "%">
             60
          <UpperBound>
       </high>
       <low type = "Bandwidth">
          <LowerBound unit = "KB">
             60
          </LowerBound>
          <UpperBound unit = "KB">
             100
          </UpperBound>
       </low>
       ...
    </ThresholdList>
</Client>
```

(b) Example of Resource Level Specification for the Compound Component "Client(PDA)" Shown in Figure 3 (b).

Figure 8: Example of Resource Level HQML Specifications.

The importance of the system resource level QoS specifications is two fold. First, they allow multimedia applications to utilize the OS and network QoS services (e.g., CPU scheduling and reservations, network bandwidth reservations) if they are available. To be able to commit necessary OS and network resources, the QoS Proxies must have prior knowledge of the expected traffic characteristics associated with each component and link before resource guarantees can be met. In Figure 8 (a), the end-to-end network QoS parameters are specified for the link between the live media server and client. The “throughput”, “delay”, “loss rate” and “jitter” here refer to the network level parameters and have different meanings from those in the application level specifications. For example, the “delay” in the network level means the interval between two TCP or UDP packets. However, in the application level, the “delay” means the interval between two *application samples*. The <Level> tag is used to specify the degree of end-to-end resource commitment required (e.g., hard (deterministic), firm (predictive), and soft (best effort)). HQML also provides tags like <CPU>, <Memory>, <Disk> to specify the QoS parameters of the end system resources. The

QoS specifications at this level are often based on a statistical model and expressed in average and deviation values. In Figure 8 (b), the end system resource QoS requirements are specified for the atomic component "prefetcher" in the client compound component.

Second, the system resource level QoS specifications set the threshold values for the linguistic values (e.g., high, low) of different resources used in adaptation and reconfiguration rules. Those threshold values actually determine the activation timing of a specific adaptation or reconfiguration action. HQML provides the <ThresholdList> tag for this purpose. In Figure 8 (a), an example of threshold list specifications are given for the linguistic values used in the reconfiguration rules of the *Live Media Streaming* application. In Figure 8 (b), the threshold specifications are given for the adaptation rules of the "client" compound component.

As we mentioned in section 1, although HQML can be used very easily to specify application-specific QoS requirements and policies, several critical issues require thorough explorations for the success of HQML, most notably *Consistency Check* and *Automatic QoS Mappings*. We have developed the visual QoS programming environment to assist application developers to create accurate HQML specifications easily.

3 Visual QoS Programming Environment *QoSTalk*

In this section, we introduce the visual QoS programming environment *QoSTalk* in detail. *QoSTalk* provides visual tools to help the application developer to create QoS specifications in HQML easily. It provides the *consistency check* on QoS specifications based on the theory of graph grammar. Further, *QoSTalk* includes the *distributed QoS compiler* to perform the automatic mappings between application and resource level QoS parameters to relieve the application developer of the burden of dealing with low level QoS specifications

3.1 Architecture Overview

The overall architecture of *QoSTalk* is shown in Figure 9. The application developer first uses the *Visual Hierarchical QoS Editor* to draw all possible application configurations, using visual tools, and input all kinds of user-level and application-level QoS requirements via dialogs. Second, the developer uses our *Consistency Check* tools to "debug" the input visual QoS specifications (application configurations with user-level QoS parameters and application-level QoS parameters for each individual components). If there is any inconsistency, the error messages are returned to the application developer in the *Visual Hierarchical QoS Editor*. Otherwise, the legal application configurations are passed to the *Distributed QoS Compiler* [43] to instrument the application source code with middleware APIs, probe the resource requirements and establish the mappings between application-level and resource-level QoS parameters automatically. In the fourth step, the legal application configurations with complete QoS specifications (user, application and resource level) are passed to the *HQML Generator*. It "traverses" the complete application configurations to

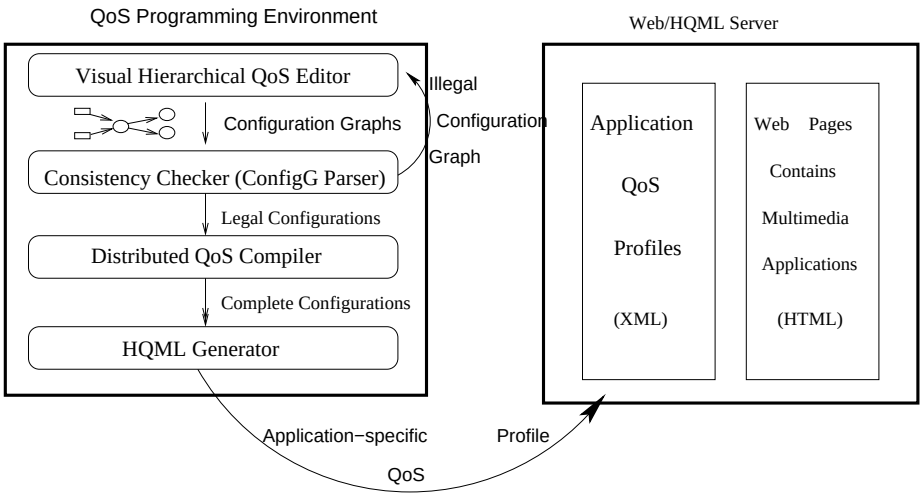
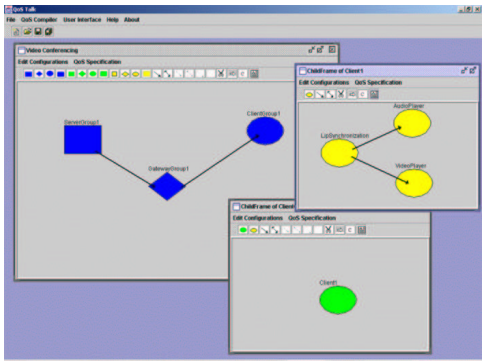


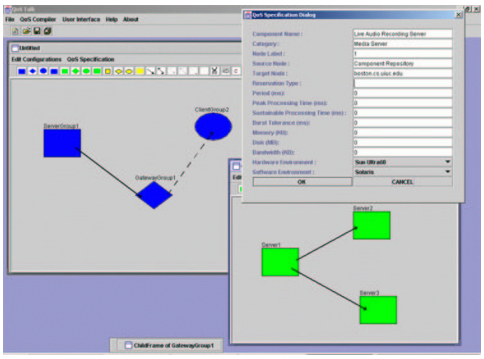
Figure 9: The QoS Programming Environment Architecture.

generate the *HQML* file automatically. Finally, the complete *HQML* file is saved into the *QoS Profile Database*.

3.2 Visual Hierarchical QoS Editor



(a) Screenshot (1)



(b) Screenshot (2)

Figure 10: Screenshots of the Visual Hierarchical QoS Editor.

The *Visual Hierarchical QoS Editor* is a visual programming tool, which allows application developers to draw application configurations for distributed multimedia applications easily. It also provides dialogs for application developers to input all kinds of QoS parameters and policies. The design of the *Visual QoS Editor* is based on the application model introduced in Section 2.1 and follows the hierarchical approach. The application developer could refine a compound component by drawing all of its subcomponents in a subframe. We use different shapes to differentiate between different component types: (1) the *rectangle* represents the *server* component; (2) the *rhombus* represents the *gateway* component; (3) the *oval* represents the *client* component; and (4) the *round rectangle* represents

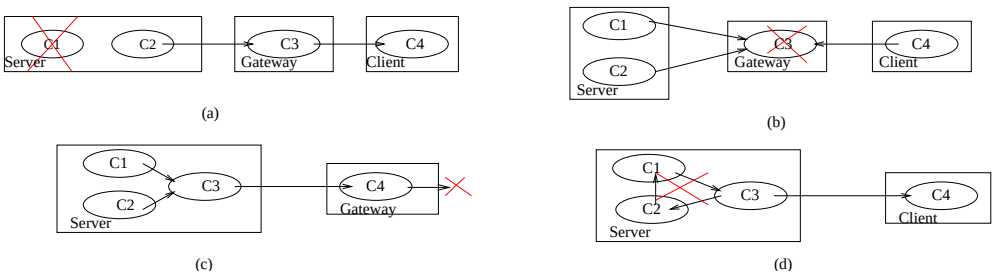


Figure 11: Illegal Configuration Graph Examples.

the *peer* component. We also use three line types to represent three different links: (1) the *solid* line is used to define the *fixed link*; (2) the *dashed* line is used to define the *mobile host link*; and (3) the *dotted* line is used to represent the *mobile user link*. Finally we use three different colors, blue, green, and yellow to represent the cluster, compound and atomic component respectively. Figure 10 shows some screenshots of the *Visual Hierarchical QoS Editor*. After the application developer finishes those visual QoS specifications, the next challenges are how to check their consistencies and generate the HQML file from them automatically. Our solution for the above challenges is to utilize the *formal graph grammar* theory. We have designed a special Boundary Symbol Relation grammar, called *ConfigG*, which will be introduced in the next section.

3.3 ConfigG: A Special Boundary Symbol Relation Grammar

The tasks of consistency check are two fold: (1) find illegal application configurations; (2) find mismatched QoS parameters between any two connected components (atomic or compound). Figure 11 shows some examples of illegal configurations. In Figure 11 (a), the atomic component “c1” is not connected with any other components; In Figure 11 (b), the client component “c4” does not receive any data flow; In Figure 11 (c), there is not client component to receive the media data flow; In Figure 11 (d), there is a loop within the server host (compound component). The examples of mismatched QoS parameters include an MPEGII encoder connected with an H261 decoder, or a low quality video player, which could only handle up to 15 fps (frames per second), connected with a high performance video server, which sends video stream at 30 fps.

Our solution to address these challenges is to utilize formal graph grammar theory. Each application configuration is described by a graph grammar sentence. Hence, the problems of consistency check on the visual QoS specifications (application configurations with application-level QoS parameters for each individual component) is reduced to that of ”debugging” a graph grammar sentence using its syntactic and semantic parsers. The *Symbol Relation* (SR) grammar [14] is a very powerful graph grammar for handling complex graph structures. We designed a special grammar, called *ConfigG*, for the *consistency check* based on the SR grammar. The consistency check is divided into two stages. In the first stage, the *ConfigG Syntactic Parser* translates the input application configurations into a *ConfigG* sentence. If the

translation process is not successful, the configuration graph is illegal. Otherwise, a derivation tree [14] is generated for the configuration graph. In the second stage, the *ConfigG Semantic Parser* traverses the derivation tree to check the semantic consistencies according to the semantic rules associated with each derivation step.

3.3.1 Symbol Relation Grammar Overview

In this section, we give a brief overview of Symbol Relation (SR) [14] Grammar. Each sentence in an SR language is composed of a set of symbol occurrences representing visual elementary objects, which are related through a set of binary relational items. The main feature of SR grammars is that the *derivation of a sentence is performed by rewriting both symbol occurrences and relation items by means of simple context-free style rules*. The basic concepts of the SR grammar are formalized in the following definitions. We will demonstrate each concept with examples from our special *Symbol Relation* grammar, called *ConfigG*. We will present *ConfigG* in the next section based on the definitions introduced in this section.

DEFINITION 3.1 Given an alphabet T, the set of *symbol occurrences (s-items)* on T is defined as $C_T = T \times N$, where N is the set of natural numbers. For simplicity, the element (X, k) of C_T will be written as X^k.

For example, the alphabet T in *ConfigG* includes *VideoServer, Transcoder,* etc. The s-items in *ConfigG* are defined as $VideoServer^0$, $Transcoder^1$, etc.

DEFINITION 3.2 Given a set R of relation symbols, an alphabet T, and M $\subseteq C_T$ (definition 3.1), a *relational item (r-item)* on R and M has the form r(X,Y), where $r \in R$, $X \in M$, $Y \in M$, and $X \neq Y$. The set **L** of r-items is defined as **L** = {r(X,Y) | $r \in R$, $X \in M$, $Y \in M$ and $X \neq Y$ }.

For example, the set R in *ConfigG* includes {fl, mul, mhl} ("fl", "mul", "mhl" represents the fixed link, the mobile user link and the mobile host link respectively). Correspondingly, the r-items are defined as:

$fl(VideoStreaming^1, Multicast^0)$, $mhl(Transcoder^2, BitmapPlayer^3)$, etc.

The notion of a *Symbol Relation (SR) sentence* can be now formally defined based on the Definition 3.1, 3.2.

DEFINITION 3.3 Given a set R of relation symbols and an alphabet T, an *SR sentence* on R and T is a pair $< M, K >$, where M is a finite nonempty subset of C_T (definition 3.1), and K is a finite nonempty subset of L (definition 3.2) . More explicitly, the general form of a sentence is

$< M, K > = < \{X_1, ...X_\mu\},\ \{r_1(Y^1, Z^1), ...r_\sigma(Y_\sigma, Z_\sigma)\} >$, where

$\mu \geq 1, \sigma \geq 1$, $X_j \in C_T$, for 1 $\leq$ j $\leq \mu$

$r_i \in R$, for $1 \leq$ i $\leq \sigma$, $Y_i, Z_i \in$ M, for $1 \leq$ i $\leq \sigma$

For example, a *ConfigG* sentence has the form as following:

$<\{VideoServer^0, VideoPlayer^0\}, \{fl(VideoServer^0, VideoPlayer^0)\}>$.

An SR grammar is specified by a set of productions that state how to rewrite s-items and r-items. The rewrite rules of s-items are called *s-productions*, and the rewrite rules of r-items are called *r-productions*. The right-hand side of each s-production consists of an SR sentence and the r-productions allow us to embed the righthand side of the applied s-production into the host sentence. In other words, r-productions replace r-items containing the rewritten s-item, with r-items relating the new s-items to existing ones in the host sentence. The above ideas are formalized in the following definition.

DEFINITION 3.4 A *Symbol Relation (SR)* grammar is a 6-tuple G = (V_N, T, R, S, SP, RP), where

- V_N is a finite nonempty set of nonterminal symbols.

- T is a finite nonempty set of terminal symbols, i.e. the alphabet in the Definition 3.1.

- R is a finite set of relation symbols, i.e. the set "R" in the Definition 3.2.

- $S \in V_N$ is the start symbol.

- SP is a finite set of symbol-item rewriting rules, called *s-productions*, of the form

$l : Y^0 \rightarrow < M, K >$, where:

l is an integer uniquely labeling the s-production; $< M, K >$ is an SR sentence on R and $V_N \cup T$; $Y \in V_N, Y^0 \notin M$.

- RP is a finite set of relation-item rewriting rules, called *r-productions*, of the form

$r(Y^0, X^1) \rightarrow [l]\, Q$ or $r(X^1, Y^0) \rightarrow [l]\, Q$, where:

$r \in R$; l is the label of an s-production $Y^0 \rightarrow < M, K >$; $X \in V_N \cup T$ and $X^1 \notin M$; $Q \neq \emptyset$ is a finite set of r-items of the form $r(Z, X^1)$ or $r(X^1, Z)$, with $Z \in M$.

The label "*l*" in the right-hand side of an r-production establishes an operative link among s-productions and r-productions; i.e., an r-production

$r(Y^0, X^1) \rightarrow [l]\, Q$.

can be applied only after the symbol Y^0 has been rewritten using the s-production with label *"l"*. More precisely, during a derivation step, a symbol occurrence X^0 in a sentence S^1 is replaced by a sentence S^2, according to a s-production of the form $X^0 \rightarrow S^2$. After X^0 has been rewritten, the replacement of the set of r-items involving X^0 is performed through r-production of the form $r(X^0, Y^1) \rightarrow Q$, where Q is a set of r-items relating Y^1 to s-items in S^2. Multiple occurrence of the same symbol within a sentence is possible. Thus, they are distinguished by different index numbers.

We conclude this section with a note about complexity issues. A limit on the complexity of parsers of graph

grammars has been given by Brandenburg in the *confluence* property.[5] The confluence property means that all grammatical derivations in a language are independent of the rewriting order of the nonterminals. This property is indispensable for efficient parsers since any order of application rules must result in the recognition of the sentences belonging to the language. The *Boundary Symbol Relation (BSR)* grammar has the confluence property and thus a lower computational complexity. An efficient parser has been given [14] for BSR grammars, which have the connectivity and limited degree properties. This last property means that the number of relations that associate one object with another is limited. Since our legal configuration graphs have the connectivity and limited degree properties, the *ConfigG*, which is used to generate all legal configuration graphs, has the confluence property and thus belongs to the *BSR* grammars.

3.3.2 ConfigG Syntactic Parser

We now introduce the special SR grammar ConfigG based on the definition 3.4 in the previous section.

ConfigG = (V_N, T, R, S, SP, RP), where

- V_N = {S, SC, GWC, CC, GC, A, B, C, D, AT, BT, CT, DT}. S is the start symbol. SC, GWC, CC and GC represent the server, gateway, client, and general composite component group (cluster) respectively. A, B, C and D represent the server, gateway, client and general composite component, respectively. AT, BT, CT and DT are temporary symbols used during derivation to avoid ambiguity;

- T = {AR, AP, VR, VP, RC, ..., etc} is the set of terminals representing the atomic components, such as AudioRecorder (AR), AudioPlayer (AP), VideoRecorder (VR), RemoteController (RC), etc;

- R = {fl, mul, mhl} is the set of terminals representing different links (fixed link, mobile user link, mobile host link) between components;

- S is the start symbol;

- SP is the set of s-productions;

- RP is the set of r-productions.

The complete set of the *ConfigG* production rules is given in [19]. A simple *ConfigG Syntactic Parser* algorithm is given in Figure 12. The "L" represents either FixedLink, MobileHostLink or MobileUserLink. Now, let us consider the configuration graph of the Live Video Streaming application, illustrated in Figure 3(b). We will show how derivation steps are performed to construct the *ConfigG* sentence according to the algorithm given in Figure 12.

Step1: We start from the following s-production:

4. $S^0 \rightarrow < \{SC^2, GWC^2, CC^2\}, \{fl(SC^2, GWC^2), mhl(GWC^2, CC^2)\} >$

We start from this s-production because the top-view of the application configuration is the Server-Gateway-Client

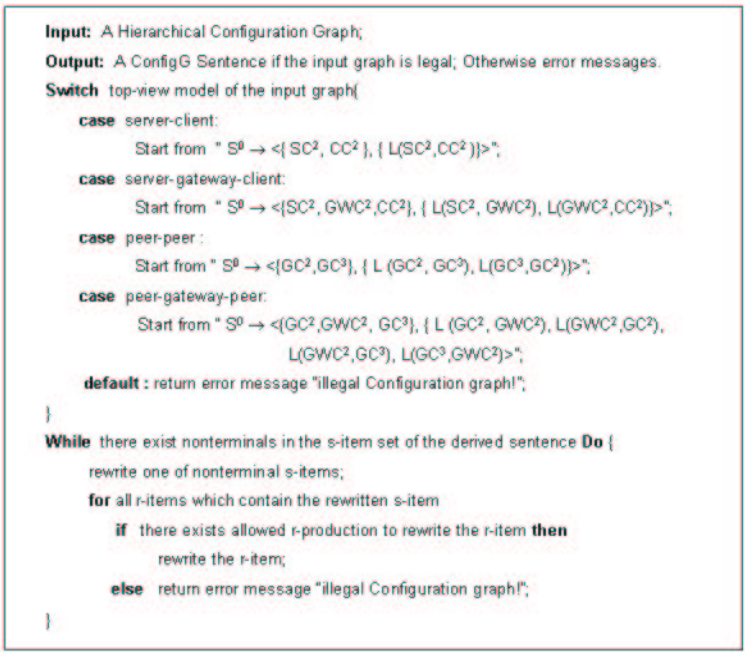


Figure 12: The ConfigG Syntactic Parser Algorithm.

model; The derived ConfigG sentence becomes the following:

$$< \{SC^2, GWC^2, CC^2\}, \{fl(SC^2, GWC^2), mhl(GWC^2, CC^2)\} > \quad (1)$$

which corresponds to the configuration graph illustrated in Figure 13(a).

Step 2: Rewrite the nonterminal s-items "SC^2", "GWC^2", and "CC^2" into "A^2", "B^2", "C^2" respectively using the following s-productions:

17. $SC^1 \rightarrow < \{A^2\}, \emptyset >$

22. $GWC^1 \rightarrow < \{B^2\}, \emptyset >$

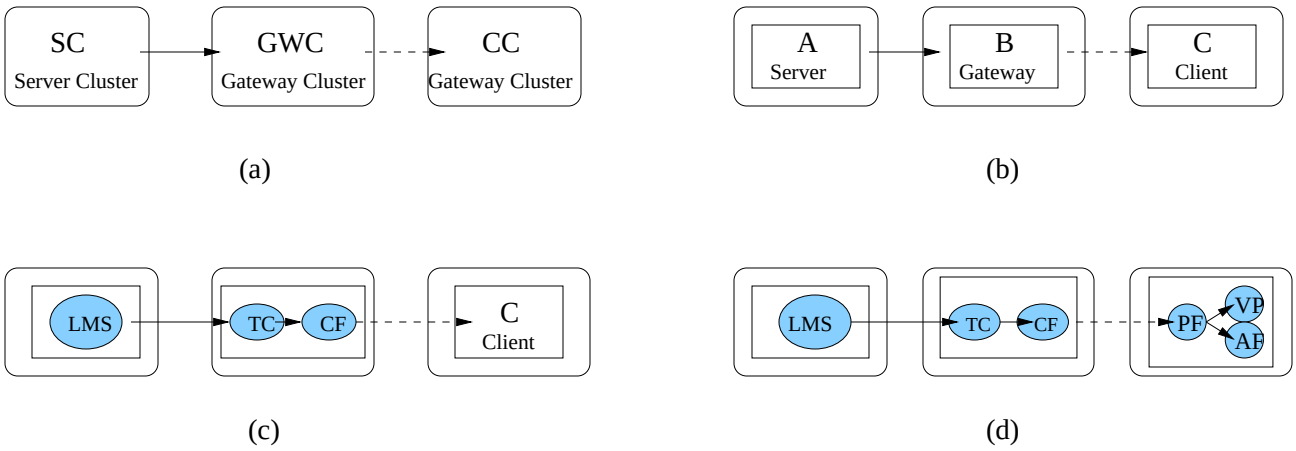
35. $CC^1 \rightarrow < \{C^2\}, \emptyset >$

Correspondingly, we need to rewrite all the r-items in the sentence (1). The derived *ConfigG* sentence becomes the following:

$$< \{A^2, B^2, C^2\}, \{fl(A^2, B^2), mhl(B^2, C^2)\} > \quad (2)$$

which corresponds to Figure 13(b) ;

Step 3: Since there is one atomic component LiveMediaServer (LMS^2), we need to rewrite the nonterminal s-item A^2 using the following s-productions:



LMS = Live Media Streaming; TC = Transcoder; CF = ColorFilter

Figure 13: ConfigG Derivation Examples.

44. $A^2 \rightarrow < \{AT^2\}, \emptyset >$

47. $AT^2 \rightarrow < \{LMS^2\}, \emptyset >$

The r-item fl(A^2,B^2) in the sentence (2), which contains the rewritten s-item A^2, must be rewritten using the following r-productions:

R112 $fl(A^2, B^2) \rightarrow [44]\{fl(AT^2, B^2)\}$

R113 $fl(AT^2, B^2) \rightarrow [47]\{fl(LMS^2, B^2)\}$

The derived *ConfigG* sentence becomes the following:

$$< \{LMS^2, B^2, C^2\}, \{fl(LMS^2, B^2), mhl(B^2, C^2)\} > \quad (3)$$

Next, because there are two atomic components Transcoder (TC^2) and ColorFilter (CF^2), we need to rewrite the nonterminal s-item B^2 using the following s-productions:

52. $B^2 \rightarrow < \{BT^2\}, \emptyset >$

55. $BT^2 \rightarrow < \{BT^3, CF^2\}, \{fl(BT^2, CF^2)\} >$

57. $BT^3 \rightarrow < \{TC^2\}, \emptyset >$ After we rewrite the related r-items correspondingly, the derived *ConfigG* sentence becomes the following:

$$< \{LMS^2, TC^2, CF^2, C^2\}, \{fl(LMS^2, TC^2), fl(TC^2, CF^2), mhl(CF^2, C^2)\} > \quad (4)$$

which corresponds to Figure 13(c);

Step 4: we rewrite the nonterminal s-items C^2 into the terminal symbols, Prefetcher(PF^2), AudioPlayer(AP^2),

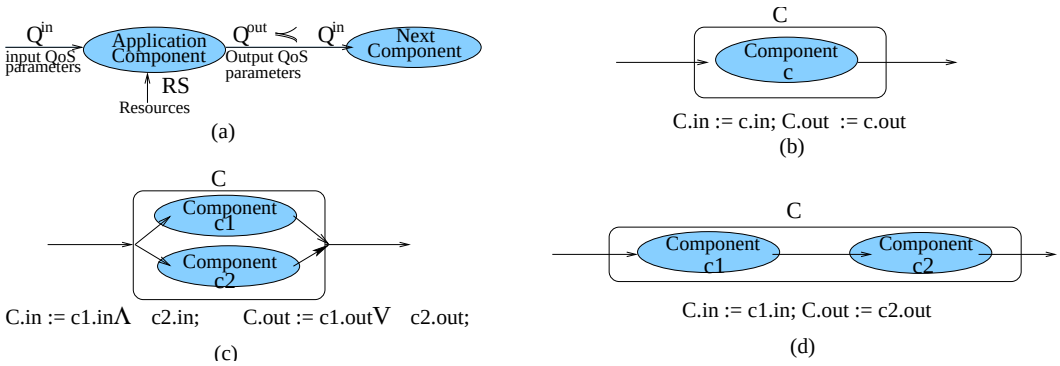


Figure 14: Categorization and Calculation of QoS parameters.

VideoPlayer(VP^2) using the following s-productions:

58. $C^2 \rightarrow < \{CT^2\}, \emptyset >$

61. $CT^2 \rightarrow < \{PF^2, CT^3\}, \{fl(PF^2, CT^3\} >$

59. $CT^3 \rightarrow < \{CT^4, CT^5\}, \emptyset >$

63. $CT^4 \rightarrow < \{VP^2\}, \emptyset >$

63. $CT^5 \rightarrow < \{AP^2\}, \emptyset >$

Then we rewrite the related r-items correspondingly, the derived sentence becomes:

$$< \{LMS^2, TC^2, CF^2, PF^2, AP^2, VP^2\}, \{fl(LMS^2, TC^2), fl(TC^2, CF^2), mhl(CF^2, PF^2), fl(PF^2, AP^2), fl(PF^2, VP^2)\} > \quad (5)$$

which corresponds to Figure 13(d);

All the illegal configurations illustrated in Figure 11 cannot be translated into any ConfigG sentence. Although *ConfigG Syntactic Parser* could check many inconsistencies in the configuration graph, it can not find all inconsistencies like mismatched QoS parameters between two connected components. Our solution is the *ConfigG Semantic Parser*, which is introduced in the next section.

3.3.3 ConfigG Semantic Parser

First, we consider a scheme of categorizing QoS parameters. For this purpose, we focus on a single application component, which could be an atomic component, a compound component or a cluster. We assume that the application component accepts input with a QoS level Q^{in} and generates output with a QoS level Q^{out}, both of which are *vectors of application-level QoS parameter values*. The value of an application-level QoS parameter could be the media type (text, image, audio, video, etc.), media format (MPEG, JPEG, H261, wav, mp3), resolution(1600*1200), frame-rate range ([15,30]), or color depth (16), etc. In order to process input and generate output, a specific amount of resources

RS is required, which is a vector of resource requirements. Figure 14(a) illustrates such characterization in terms of QoS parameters and resources. Formally, we define the vectors Q^{in}, Q^{out}, and *RS* as follows:

$$\begin{aligned} Q^{in} &= [q_1^{in}, q_2^{in}, ..., q_n^{in}] \\ Q^{out} &= [q_1^{out}, q_2^{out}, ..., q_n^{out}] \\ RS &= [RS_1, RS_2, ..., RS_m] \end{aligned} \tag{6}$$

Intuitively, if a component A is connected to a component B, the output QoS of A (Q_A^{out}) must "match" the input QoS requirements of component B (Q_B^{in}). In order to formally describe this *QoS consistency* requirements, we define an inter-component relation "$\preceq$", called *Satisfy*, as follows: $Q_A^{out} \preceq Q_B^{in}$ if and only if

$$\begin{aligned} \forall i, 1 \le i \le Dim(Q_B^{in}),\ \exists j, 1 \le j \le Dim(Q_A^{in}), \quad & q_{Aj}^{out} = q_{Bi}^{in},\ \ if\ q_{Bi}^{in}\ is\ a\ single\ value; \\ & q_{Aj}^{out} \subseteq q_{Bi}^{in},\ \ if\ q_{Bi}^{in}\ is\ a\ range. \end{aligned} \tag{7}$$

The Dim(Q_A) represents the dimension of the vector "Q_A". The single value QoS parameters include media type (text, image, video, etc.), media format (JPEG, MPEG, etc.), resolution (1024*768), etc. The range value QoS parameters include throughput ([10fps,30fps]), jitter ([0,3ms]), delay ([0,300ms]), lossrate ([0,20%]).

Based on the above definitions, we further define how to calculate QoS parameter vector of a compound component from its subcomponents' QoS parameter vectors. We define a function "$\bigwedge$" between two QoS parameter vectors, called $intersection^v$, as following: $Q_C = Q_A \bigwedge Q_B$ if and only if

$$\begin{aligned} & S_C = S_A \cup S_B \\ \forall k,\ 1 \le k \le card(S_C),\ \ if\ s_k \in S_A \cap S_B\ \ and\ \ q_{Ai}.type = q_{Bj}.type = s_k,\ \ q_{Ck} = q_{Ai} \cap q_{Bj} \\ & if\ s_k \in S_A \cap \overline{S_B}\ \ and\ \ q_{Ai}.type = s_k,\ \ q_{Ck} = q_{Ai} \\ & if s_k \in \overline{S_A} \cap S_B\ \ and\ \ q_{Bj}.type = s_k,\ \ q_{Ck} = q_{Bj} \end{aligned} \tag{8}$$

S_A, S_B, S_C represent the sets of the QoS parameter types contained in Q_A, Q_B and Q_C respectively. (For example, S_A = { $1 \le i \le Dim(Q_A) \mid Q_A.type$ }). The definition "card(S_C)" represents the number of elements in the set S_C. We also define a new function "$\bigvee$" between two QoS parameter vectors, called $union^v$, as following: $Q_C = Q_A \bigvee Q_B$ if and only if

$$\begin{aligned} & S_C = S_A \cup S_B \\ \forall k,\ 1 \le k \le card(S_C),\ \ if\ s_k \in S_A \cap S_B\ \ and\ \ q_{Ai}.type = q_{Bj}.type = s_k,\ \ q_{Ck} = q_{Ai} \cup q_{Bj} \end{aligned}$$

Table 1: Semantic rules of s-productions for Consistency Check

(4). $S^0 \rightarrow < \{SC^2, GWC^2, CC^2\}, \{fl(SC^2, GWC^2), fl(GWC^2, CC^2)\} >$ $\{ SC^2.out \preceq GWC^2; GWC^2 \preceq CC^2; SC^0.in := SC^2.in; SC^0.out := CC^2.out.\}$
(13) $SC^0 \rightarrow < \{SC^2, SC^3\}, \emptyset >$ $\{ SC^0.in := SC^2.in \bigwedge SC^3.in; SC^2.out := SC^2.out \bigvee SC^3.out. \}$
(47) $AT^0 \rightarrow < \{AT^2, \alpha^2\}, \{fl(AT^2, \alpha^2)\} >$ $\{ AT^2.out \preceq \alpha^2; AT^0.in := AT^2.in; AT^0.out := \alpha^2.out. \}$

$$if\ s_k \in S_A \cap \overline{S_B}\ \ and\ \ q_{Ai}.type = s_k,\ \ q_{Ck} = q_{Ai}$$
$$if\ s_k \in \overline{S_A} \cap S_B\ \ and\ \ q_{Bj}.type = s_k,\ \ q_{Ck} = q_{Bj} \quad (9)$$

We use attribute "in" to represent an application component's input QoS parameter vector, and the attribute "out" to represent its output QoS. If the subgraph of the compound component is like the one illustrated in Figure 14(b), the compound component's attributes, "in" and "out", are the same as those of its subcomponent. If the subgraph is like the one illustrated in Figure 14(c), the attribute "in" of the compound component is the $intersection^v$ of the two subcomponents' "in" attributes. Intuitively, it means that the input QoS parameters of the compound component must *satisfy* both subcomponents' input QoS parameters, because the stream is sent to both components. The attribute "out" of the compound component is the $union^v$ of the two subcomponents' "out" attributes. It means that the next component, connected to this compound component, must be able to handle both subcomponents' output QoS parameters. If the configuration of the compound component is like the one illustrated in Figure 14(d), the attribute "in" of the compound component is the same as that of its first subcomponent. The attribute "out" of the compound component is the same as that of its second subcomponent. In order to use *ConfigG Semantic Parser* to perform consistency check, we associate semantic rules with each s-production (semantic rules for r-productions are not needed for this purpose). We also assign attributes "in" and "out" to each s-item as mentioned above. Some examples of semantic rules for s-productions are given in table 1. The first rule in table 1 says that if the s-production with label 4 is used in a derivation step, the output QoS of SC^2 must *satisfy* the input QoS of GWC^2 because SC^2 is connected with GWC^2 by a FixedLink "fl(SC^2,GWC^2)". Similarly, the output QoS of GWC^2 must also *satisfy* the input QoS of CC^2. The rule also specifies that the attributes "in" and "out" of the s-item S^0 are calculated by assigning "SC^2.in" and "CC^2.out" to them respectively. The other rules can be interpreted similarly.

If *ConfigG Syntactic Parser* derives a *ConfigG* sentence from an input application configuration successfully, a *derivation tree* is generated. For example, Figure 15 shows the derivation tree of the *ConfigG* sentence (4) for the *Live Media Streaming* application, illustrated in figure 13(c). Then *ConfigG Semantic Parser* traverses the derivation

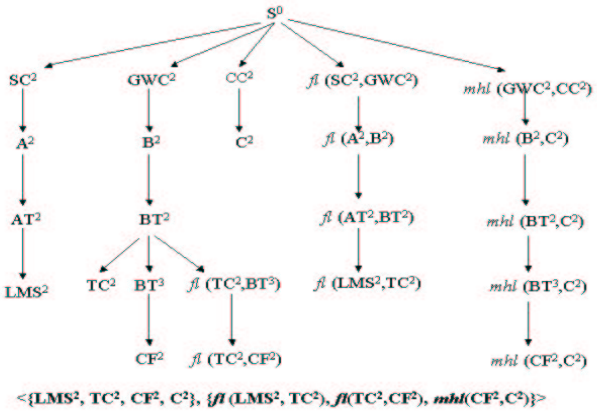


Figure 15: Example of ConfigG Derivation Tree for the Application Configuration Shown in Figure 13 (c).

Table 2: Semantic rules of s-productions for HQML file generation

---

(4) $S^0 \rightarrow < \{SC^2, GWC^2, CC^2\}, \{fl(SC^2, GWC^2), fl(GWC^2, CC^2)\} >$
$S^0.hqml :=$ " $< AppConfig >''$ || " $< ServerCluster >''$ || $SC^2.hqml$ || " $< /ServerCluster >''$
|| " $< GatewayCluster >''$ || $CWC^2.hqml$ || " $< /GatewayCluster >''$ || " $< ClientCluster >''$
|| $CC^2.hqml$ || " $< /ClientCluster >''$ || " $< LinkList >''$ || " $< Link >''$ || $fl.hqml$ || " $< /Link >''$
|| " $< Link >''$ || $fl.hqml$ || " $< /Link >''$ || " $< /LinkList >''$ || " $< /AppConfig >$"

(13) $SC^0 \rightarrow < \{SC^2, SC^3\}, \emptyset >$
$SC^0.hqml := SC^2.hqml$ || $SC^3.hqml$

---

tree depth-first to check the consistency ("$\preceq$" operation) and calculates the attributes for the left-hand side s-item, according to the semantic rules associated with each s-production used in each derivation step. If any inconsistency is found, *ConfigG Semantic Parser* stops traversing the derivation tree immediately, and sends error messages back to the *Visual Hierarchical QoS Editor*. Otherwise, the consistency check is successful for the input appliation configuration. A byproduct of this procedure is the end-to-end application-level QoS input (S^0.in) and QoS output (S^0.out) for the distributed multimedia application. If we associate another attribute "s" with each s-item to represent its system resource requirements, the end-to-end resource requirements for a particular application configuration could also be derived in a straightforward manner.

The *ConfigG Semantic Parser* can also be used to generate HQML file automatically. Each s-item or r-item is assigned the attribute "hqml", which represents the HQML file fragment the s-item or r-item generates. The *ConfigG Semantic Parser* traverses the *ConfigG derivation tree* of the *ConfigG sentence* for an application configuration with *complete* and *consistent* QoS specifications, generating the HQML file according to the semantic rules associated with each s- or r- productions used in each derivation step. Some examples of the semantic rules for s-productions are given

in table 2, where the symbol "||" denotes the string concatenation function.

3.4 Distributed QoS Compiler

Application developers could specify the system resource requirements in the *Visual Hierarchical QoS Editor* using dialogs easily if they know them in advance. However, it is often difficult for developers to map the application level QoS parameters into the system resource level ones directly. Thus, we propose the *Distributed QoS Compiler*, which is part of the *QoSTalk* framework and performs the *Automatic QoS Mappings*, for the developer, from the application level QoS parameters into the system resource level parameters. The automatic translation is based on two major approaches: (1) the *analytical translation*; and (2) the *probing and profiling* services.

The *analytical translation* is based on *translation functions* between application level QoS parameters and system resource parameters. Examples of the analytical translation functions from the application QoS parameters to the transport subsystem QoS parameters are $P_N = \lceil M_A/M_N \rceil$ x P_A, and $B_N = P_N$ x M_N where P_N, M_A, M_N, P_A, and B_N are transport packet rate, application sample size, transport packet size, application sample rate, and transport network bandwidth, respectively. M_A and P_A can be frame size and frame rate specified as application level QoS parameters. Another example is the translation of MPEG video's QoS parameters into CPU requirements, which is presented in [24].

While the analytical translation gives precise resource requirements for some cases, the dependencies among different QoS parameters, and the dependencies among multiple system resources make an accurate translation function hard to be quantified. To predict reasonable resource requirements in these cases, we utilize resource *probing and profiling* techniques with the optimistic assumption that the probing result is the minimum resource requirements for instantiating an application configuration.

The *probing and profiling* services are based on the *distributed probing protocol* [28] [43]. Based on the application level QoS specifications of an application configuration, the QoS compiler collaborates with the distributed runtime systems to dynamically download, start, and stop an application configuration in a distributed lightly loaded environment. The system resource requirements (e.g., different threshold values) are discovered and predicted by multiple resource brokers, notably cpu, network bandwidth and power. A hierarchical QoS probing algorithm has been given in [28].

4 HQML Executor

In this section, we introduce the *HQML executor* module, which translates the HQML specifications into desired data structures and cooperates with the QoS Proxies to provide QoS enabled distributed Web multimedia applications. Figure 16 illustrates the overall architecture of the QoS enabled Web browser, the QoS Proxies and the underlying

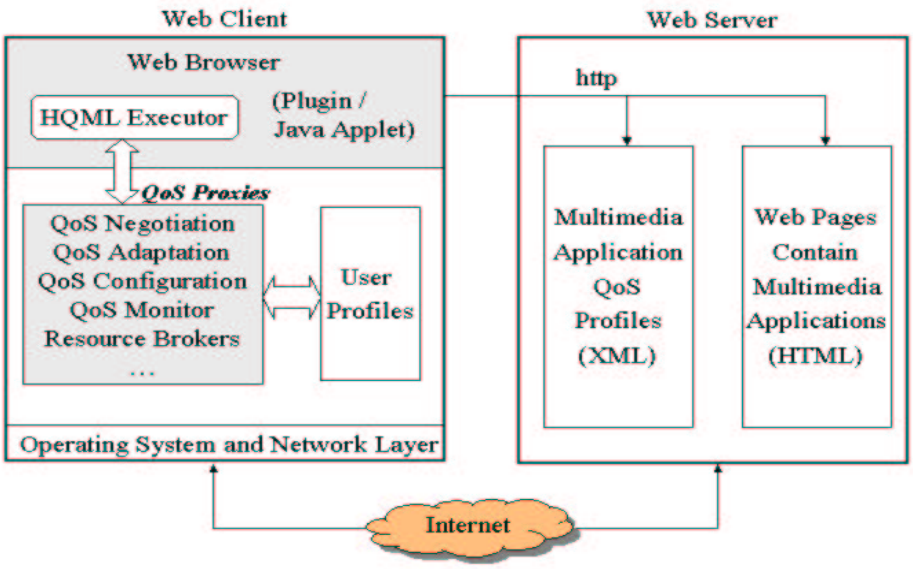


Figure 16: The QoS Enabled Web Brower Architecture.

OS and network systems. The major steps for the HQML executor to carry out, during the runtime phase, are the following:

- **Step 1:** The HQML executor intercepts the user's request for the distributed Web multimedia application "X", his/her focus of attention and desired QoS level. It then contacts the local QoS Proxies, such as Resource Brokers and Monitors, to get client's current resource availabilities (e.g., CPU, bandwidth, memory, disk, power).

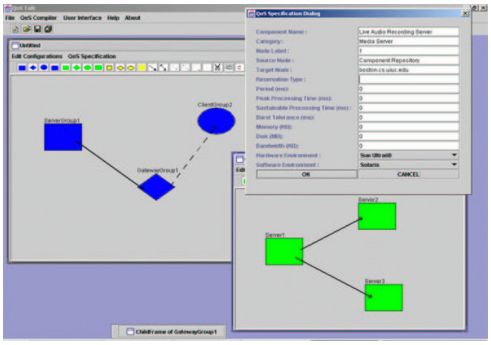
- **Step 2:** The HQML executor forwards the user QoS requests with client's resource conditions and also its platform information (e.g., PDA, Laptop, or Desktop with Windows CE, PalmOS or windows 2000) to the Web/HQML server. The Web/HQML server searches the related HQML files (QoS Profiles) and finds a set of possible application configurations that matches the user's requests and could also be supported by the client's current resource availabilities. The match is found based on the user level and system resource level QoS specifications in HQML files. The Web/HQML server then sends those HQML files back to the HQML Executor or failure message if no match could be found.

- **Step 3:** The HQML executor displays all possible choices to the user according to the received HQML files. The user could choose one of them according to their prices and his/her preferences for different service providers. Then the HQML executor translates the chosen HQML specification file (Application level and system resource level) into desired data structures by the underlying QoS Proxies. For example, the HQML executor retrieves adaptation rules and feeds them into the *QoS adaptor*; It retrieves the application configuration, reconfiguration rules and sends them to the *QoS configurator*; It may also get the system resource requirements and feeds them

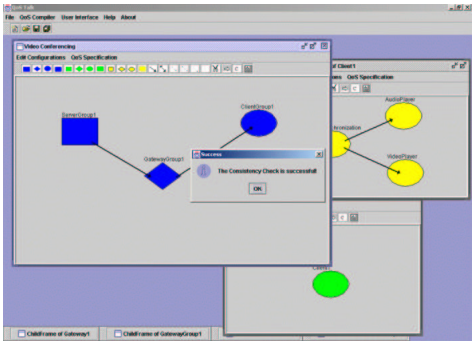
into different *resource brokers*. The *resource brokers* initiate resource reservations for the application if the underlying OS and network QoS services are available.

After the above three steps, the QoS Proxies will be responsible for providing the user with QoS enabled distributed Web multimedia applications. The QoS Proxies query the user profile to personalize the adaptation and reconfiguration rules. They collaborate with other QoS Proxies in the distributed environment to set up and maintain the end-to-end QoS level according to the policies and requirements they receive from the HQML executor. By using HQML, users receive satisfactory QoS from distributed multimedia applications on the Web, within the end-to-end resource constraints, automatically.

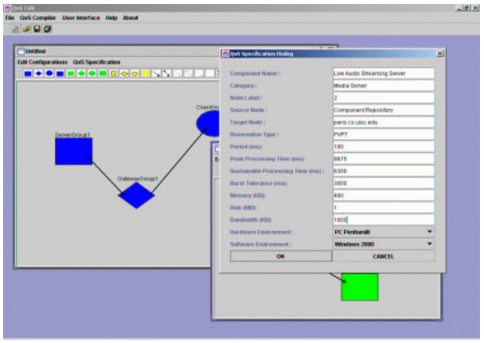
5 Implementation and Experiments



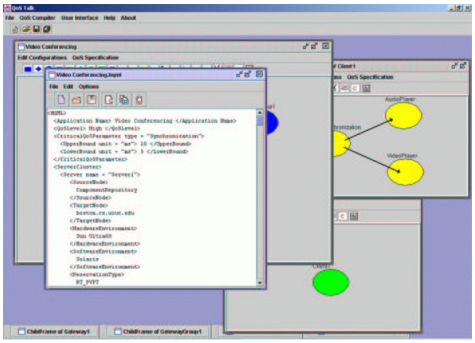
Step1: Draw Hierarchical Application Configurations&
Input Application Level QoS parameters and Policies
(a)



Step2: Consistency Check
(b)



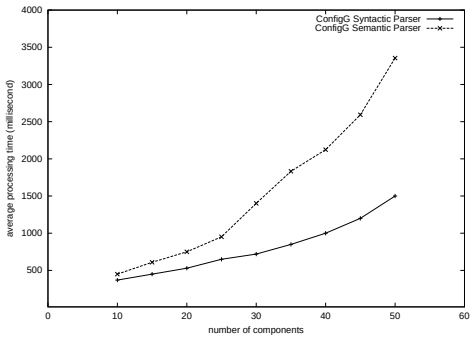
Step3: Invoke Distributed QoS Compiler to derive the system resource requirements in the Editor
(c)



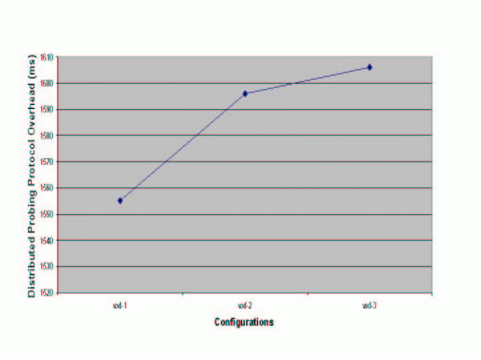
Step4: Generate the HQML specification file
(d)

Figure 17: The Workflow in the Visual QoS Programming Environment QoSTalk.

We have implemented a prototype of QoSTalk on top of the $2K^Q$ middleware system. [44] The *Hierarchical Visual QoS Editor* is implemented in Java Swing. The *ConfigG Syntactic Parser*, *ConfigG Semantic Parser*, *Distributed QoS Compiler* and *HQML Executor* are also implemented in Java. The QoSProxies, namely the unified middleware



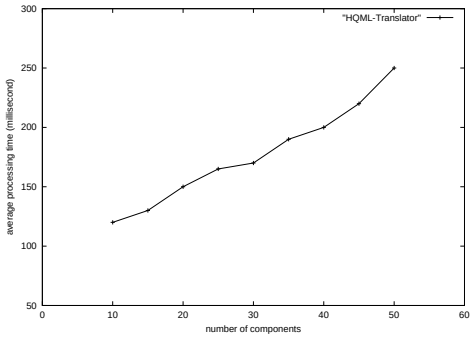
(a) Average Processing Time of ConfigG Syntactic Semantic Parser



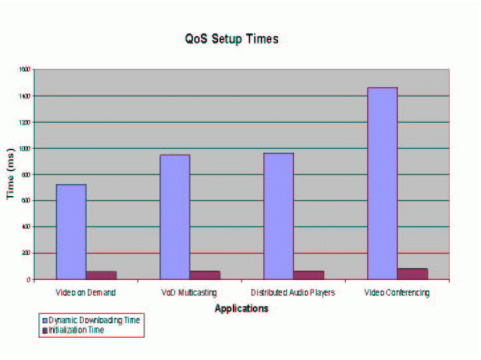
(b) Average Processing Overhead of Distributed QoS Compiler

Figure 18: Average Processing Time of QoS Programming Phase.

management entities, such as Configurator, Adaptor, and Resource Brokers, are implemented as CORBA objects and written in C++. The multimedia application components are also implemented as CORBA objects. Figure 17 illustrates the workflow of building a QoS-aware *Video On Demand* application in *QoSTalk* framework. It includes both the off-line programming phase and the runtime instantiation phase. Figure 18 (a) shows the average processing time of *ConfigG Syntactic Parser* and *Semantic Parser* for different application configurations. The x-axis represents the number of components (including links) contained in the application configuration graphs. The processing times (y-axis) of both parsers increase polynomially with the number of components. Figure 18 (b) shows the overhead of probing protocol in the Distributed QoS Compiler for three different configurations of the Video On Demand application.



(a) Average Processing Overhead of HQML Translation



(b) Average Overhead of QoS Setup using QoS Proxies in LAN

Figure 19: Average Processing Overhead of Runtime Phase.

Figure 19 shows the average processing overhead of runtime instantiation phase. It includes two parts: (a)

The *HQML* translation overhead increases linearly with the number of components (including links) contained in application configurations, illustrated by Figure 19(a). (b) Runtime instantiation overhead includes the dynamic downloading time of components from the component repository and initialization overhead, illustrated in Figure 19(b).

6 Related Work

In the multimedia language research community, several new languages have been proposed to address the challenges of supporting distributed multimedia applications on the WWW. Most notably TAOML [7], an extension of HTML, has been introduced to allow distributed multimedia applications to be prototyped on the Web easily. TAOML framework is based on the concept of Teleaction Object, which is a multimedia object with associated hyper-graph structure and knowledge structure. TAOML can also be translated into XML. But TAOML mainly focuses on the integration and synchronization issues in prototyping distributed multimedia applications. Their support for QoS is limited to the QoS services of network subsystems. Thus TAOML cannot utilize many other available QoS services such as CPU reservation and scheduling services, middleware adaptation and dynamic reconfiguration services. Moreover, application developers are required to explicitly deal with the system resource QoS parameters (e.g., bandwidth), which often cannot be mapped from application-level QoS parameters (e.g., frame rate) straightforwardly. The quality of service is also considered in the presentation layer of the multimedia systems [18, 32]. In [31], authors propose an extension of HTML to describe the meta-data for using QoS management in the WWW. However, their specification language only provides limited QoS support for simple multimedia applications because of the limitation of HTML. Recently, researchers have proposed new formatting standards like XML to address the limitations of HTML. XML has been used as user interface language [33, 23], application description language [13] and many other specification languages due to its extensibility and flexibility. Our work is orthogonal and complementary to the above approaches, since our research focuses on leveraging XML and all the state-of-the-art QoS technology to provide access to QoS support for *complex distributed multimedia applications* on the WWW in the heterogeneous environments.

In the QoS research community, several recent works have addressed the problems of QoS specifications from different directions and at different levels, namely user level, application level and system resource level. In INDEX project [1], authors address the user-level QoS specifications, such as different user preferences and price models, in detail. Some system resource level QoS specifications such as RSL by the Globus project [11, 16] are also developed. However, much effort has been put in how to specify QoS at the application level. In [37], a scripting language SafeTcl is implemented to allow existing applications, written in C language, to take advantage of QoS facilities described by the DiffServ framework. In QOS-A [40], 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

	User Level QoS	Application Level QoS	Resource Level QoS	Automatic Mapping	Independent Language	Integration with Web Application	Consistency Check	Automatic Specification Generation
Index	Yes	No	No	No	No	No	No	No
SafeTcl	No	Yes	No	No	No	No	No	No
QoS−A	No	Yes	No	No	No	No	No	No
QuAL	No	Yes	Yes	No	No	No	No	No
QDL	No	Yes	Yes	No	Yes	No	No	No
QML	No	Yes	No	No	Yes	No	No	No
RSL	No	No	Yes	No	Yes	No	No	No
TAOML	No	No	Yes	Yes	Yes	Yes	No	No
HQML	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Figure 20: Comparisons among Different QoS Specifications.

C data structure, including all the conceived clauses. In [15], authors developed QuAL (Quality-of-Service Assurance Language), which is a process-oriented programming language and further extends C language. 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. Moreover, all these approaches are tightly coupled with C programming language. Thus it is difficult for applications written in other languages like Java to utilize them. The Quality Object (QuO) framework [30, 2, 36] supports QoS at the CORBA object layer by opening up distributed object implementations to give access to the system properties of the CORBA ORB and objects. QuO extends the CORBA functional IDL (Interface Definition Language) with a QoS Description Language (QDL). QDL allows specifications of possible QoS states, the system resources and mechanisms for measuring and providing QoS, and behavior for adapting to changes in QoS. QML (QoS Modeling Language)[39] is another independent QoS specification language for distributed object systems (by independent, we mean the specification language is not coupled with any specific programming language). It allows users to specify non-functional aspects of services (such as QoS specifications) separate from the interface definition. However, QDL and QML does not consider the QoS specifications about multiple possible configurations for the reconfigurable applications.[45] Furthermore, they does not provide any *consistency check* mechanism to prevent incorrect QoS specifications, such as illegal configurations or mismatched QoS requirements, from injecting into the underlying systems. Most of all, they are not extensible and cannot be used by Web applications conveniently. In Figure 20, we summarize the main features of the QoS specification languages and multimedia languages introduced above.

7 Conclusions

In this paper, we introduce an XML-based QoS Enabling language for the Web, called *HQML*, an acronym for "Hierarchical QoS Markup Language". HQML allows different distributed multimedia applications including all the legacy applications on the WWW, to utilize all kinds of available QoS technology (middleware, OS and network). Then, we introduce a visual QoS programming environment, called *QoSTalk*, which assists application developers to create HQML files correctly and easily. We propose a special boundary symbol relation grammar, called *ConfigG*, to perform *consistency check* and generate HQML files automatically. Finally, we introduce the *distributed QoS compiler* to perform automatic QoS mappings between different levels and thus relieve the application developer of the burdens of dealing with complex low level QoS specifications. The future works include extending HQML and visual QoS programming environment to meet the following challenges: (1) *scalability* (how do we apply HQML to complex distributed applications with a large amount of components and many configurations; (2) *extensibility* (how do we include new QoS services, such as security and power management, easily.) and (3) *reusability* (how do we apply HQML to different types of multimedia applications.).

8 Acknowledgment

This work was supported by the NASA grant under contract number NASA NAG 2-1406, National Science Foundation under contract number 9870736, National Science Foundation Career Grant under contract number NSF CCR 96-23867, NSF CISE Infrastructure grant under contract number NSF EIA 99-72884, NSF CISE Infrastructure grant under contract number NSF CDA 96-24396. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NASA, NSF, or the U.S. government.

References

[1] J. Altmann and P. Varaiya. INDEX: User Support for Buying QoS with Regard to User's Preferences. *Sixth International Workshop on Quality of Service (IWQOS98)*, May 1998.

[2] John A.Zinky, David E.Bakken, and Richard D.Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*,, 1997.

[3] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski. An Application Model for Pervasive Computing. *Proceedings of MobiCOM 2000: the 6th Annual International Conference on Mobile Computing and Networking*, 2000.

[4] Gordon S. Blair, Geoff Coulson, Nigel Davies, Philippe Robin, and Tom Fitzpatrick. Adaptive Middleware for Mobile Multimedia Applications. *Network and Operating System Support for Digital Audio and Video*, 1997.

[5] F.J. Brandenburg. On polynomial time graph grammars. *Lecture Notes in Computer Science,*, 294, 1988.

[6] A. Campbell and G. Coulson. A QoS Adaptive Transport System: Design, Implementation and Experience. *Forth ACM International Conference on Multimedia (ACM Multimedia 96)*, 1996.

[7] S. K. Chang. Multimedia Software Engineering. *Kluser Academic Publishers, Boston/Dordrecht/London*, 2000.

[8] Z. Chen, S. M. Tan, R. H. Campbell, and Y. Li. Real Time Video and Audio in the World Wide Web. *World Wide Web Journal, vol. 1*, 1996.

[9] World Wide Web Consortium. eXtensible Markup Language. *http://www.w3c.org/XML/*.

[10] World Wide Web Consortium. Synchronized Multimedia Integration Language (SMIL) 1.0 Specification. *http://www.w3c.org/TR/REC-smil/*.

[11] Karl Czajkowski, Ian Foster, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A Resource Management Architecture for Metacomputing Systems. *Proceedings of IPPS/SPDP98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[12] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. *http:www.w3c.org/TR/1998/NOTE-xml-ql-19980819/*, 1998.

[13] K. Eustice, T. Lehman, A. Morales, M.C. Muson, S. Edlund, and M. Guillen. A Universal Information Appliance. *IBM Systems Journal*, 1999.

[14] F. Ferrucci, G. Pacini, and G. Satta. Symbol-Relation Grammars: A Formalism for Graphical Languages. *Information and Computation*, 131, 1996.

[15] P. Florissi. QoSME: QoS Management Environment. *PhD Thesis, Department of Computer Science, Columbia University*, 1996.

[16] Ian Foster and Carl Kesselman. The Globus Project: A Status Report. *Proceedings of IPPS/SPDP98 Heterogeneous Computing Workshop*, 1998.

[17] A. Fox, S. D. Gribble, and Y. Chawathe. Adapting to Network and Client Variation Using Infrastrural Proxies: Lessons and Perspectives. *IEEE Personal Communications*, August 1998.

[18] K. Fujikawa and et.al S. Shimojo. Application Level QoS Modeling for a Distributed Multimedia System. *Proceedings of 1995 Pacific Workshop on Distributed Multimedia Systems*, March 1995.

[19] X. Gu. Visual Quality of Service Programming Environment for Distributed Heterogeneous Systems. *MS Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign*, January 2001.

[20] X. Gu and K. Nahrstedt. An Event-Driven, User-Centric, QoS-aware Middleware Framework for Ubiquitous Multimedia Applications. *Proc. of 9th ACM Multimedia (Multimedia Middleware Workshop)*, 2001.

[21] X. Gu, D. Wichadakul, and K. Nahrstedt. Visual QoS Programming Environment for Ubiquitous Multimedia Services. *Proc. of IEEE International Conference on Multimedia and Expo 2001 (ICME2001)*, 2001.

[22] A. Hafid and G. Bochmann. Quality of Service Adaptation in Distributed Multimedia Applications. *ACM Springer-Verlag Multimedia Systems Journal, vol. 6, no. 5*, 1998.

[23] Todd D. Hodes and R. H. Katz. A Document-based Framework for Internet Application Control. *2nd USENIX Symposium on Internet Technologies and Systems*, 1999.

[24] K. Kim and K. Nahrstedt. QoS Translation and Admission Control for MPEG Video. *Proceedings of IEEE/IFIP International Workshop on QoS 1997 (IWQoS 97)*, May 1997.

[25] T. Kindberg and J. Barton. A Web-based Nomadic Computing System. *Computer Networks, Special Edition on Pervasive Computing*, 2001.

[26] R. Koster and T. Kramp. Structuring QoS-Support Services with Smart Proxies. *Middleware 2000: IFIP/ACM International Conference on Distributed Systems Platforms*, 2000.

[27] C. Lee, J. Lehoczky, R. Rajkumar, and D. Siewiorek. On Quality of Service Optimization with Discrete QoS Options. *Proceedings of the IEEE Real-time Technology and Applications Symposium*, 1999.

[28] B. Li, W. Kalter, and K. Nahrstedt. A Hierarchical Quality of Service Control Architecture for Configurable Multimedia Applications. *Journal of High Speed Networks, Special Issue on Management of Multimedia Networking IOS Press*, 2001.

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

[30] Joseph P. Loyall, Richard E. Schantz, John A.Zinky, and David E.Bakken. Specifying and Measuring Quality of Service in Distributed Object Systems. *Proceedings of ISORC98*, 1998.

[31] E. Madja, A. Hafid, R. Dssouli, G. v. Bochmann, and J. Gecsei. Meta-data Modeling for Quality of Service (QoS) Management in the World Wide Web (WWW). *Proceedings of International Conference on Multimedia Modeling, Lausanne, Switzerland*, 1998.

[32] D. Maier, R. Staehli, and J. Walpole. Quality of Service Specification for Multimedia Presentations. *Multimedia Systems, 3 (5/6), pp. 251-263, Springer-Verlag*, 1995.

[33] Mozilla.org. Introduction to a XUL (XML-based User Interface Language). *http://www.mozilla.org/xpfe/xptoolkit/xulintro.html*.

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

[35] 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.

[36] Joseph P.Loyall, David E. Bakken, Richard E.Schantz, John A.Zinky, David A.karr, Rodrigo Vanegas, and Kenneth R.Anderson. QoS Aspect Languages and Their Runtime Integration. *Lecture Notes in Computer Science*, 1511, May 1998.

[37] T. Roscoe and G.Bowen. Script-driven Packet Marking for Quality of Service Support in Legacy Applicaitons. *Proceedings of SPIE Conference on Multimedia Computing and Networking 2000*, January 2000.

[38] S. Servetto, K. Ramchandran, V. Vaishampayan, and K. Nahrstedt. Multiple Description Wavelet Based Image Coding. *IEEE Transactions on Image Processing*, May 2000.

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

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

[41] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, and D. Bakken. QuO's Runtime Support for Quality of Service in Distributed Objects. *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 98), Springer*, 1998.

[42] M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communication of the ACM, 36(7), pp. 74-84*, 1993.

[43] 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*, 2001.

[44] D. Wichadakul, K. Nahrstedt, X. Gu, and D. Xu. 2KQ+: An Integrated Approach of QoS Compilation and Component-Based, Runtime Middleware for the Unified QoS Management Framework. *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, 2001.

[45] D. Xu, D. Wichadakul, and K. Nahrstedt. Multimedia Service Configuration and Reservation in Heterogeneous Environments. *Proceedings of IEEE International Conference on Distributed Computing Systems(ICDCS 2000)*, April 2000.

A Schema DTD for HQML

The document type definition (DTD) for the initial, minimal version of *user level* HQML is as follows.

< !ElEMENT App (Configuration+) >

< !ATTLIST App name CDATA #REQUIRED ServiceProvider CDATA #IMPLIED>

< !ELEMENT Configuration (UserLevelQoS, QoSPreference, Price, PriceModel)>

< !ATTLIST Configuration id ID #REQUIRED >

< !ELEMENT UserlevelQoS (#PCDATA) >

< !ELEMENT QoSPreference (#PCDATA) >

< !ELEMENT Price (#PCDATA) >

< !ATTLIST Price unit DATA #REQUIRED >

< !ELEMENT PriceModel (#PCDATA) >

The document type definition (DTD) for the initial, minimal version of *application level* and *system resource level* HQML is as follows. Due to the page limit, we omitted DTD for the tags which have the similar structure to the following tags.

< !ELEMENT AppConfig (CriticalQoS, ServerCluster*, GatewayCluster*, ClientCluster*, PeerCluster*, LinkList, ReconfigRuleList?, ThresholdList?) >

< !ATTLIST AppConfig id ID #REQUIRED >

< !ElEMENT CriticalQoS (Range) >

< !ATTLIST CriticalQoS type CDATA #REQUIRED >

< !ELEMENT Range (UpperBound,LowerBound)>

< !ATTLIST Range unit CDATA #REQUIRED >

< !ELEMENT UpperBound (#PCDATA) >

< !ELEMENT LowerBound (#PCDATA) >

< !ELEMENT ServerCluster (Server+, LinkList?) >

< !ELEMENT Server (HostAddr*, Hardware?, Software?, Atomic+, CPU?, Memory?, Disk?, Power?, LinkList?, AdaptationRuleList? >

< !ATTLIST Server type (required | replacable | optional) 'required' >

< !ELEMENT HostAddr (#PCDATA) >

< !ATTLIST HostAddr type (primary | alternative) 'alternative' >

< !ELEMENT Hardware (#PCDATA) >

< !ELEMENT software (#PCDATA) >

< !ELEMENT Atomic (name, Method*, InputQoSList, OutputQoSList, CPU?, Memory?, Disk?, Power?) >

< !ATTLIST Atomic type (required | replacable | optional) 'required' >

< !ELEMENT name (#PCDATA) >

< !ELEMENT Method (param*) >

< !ATTLIST Method name CDATA #REQUIED >

< !ELEMENT param (#PCDATA) >

< !ATTLIST param name CDATA #REQUIRED lexType (int | real | boolean | enum | string) 'string' >

< !ELEMENT InputQoSList (MediaObjectList, Delay?, Jitter?, LossRate?, FrameRate?, FrameSize?,...) >

< !ELEMENT MediaObjectList (MediaObject+) >

< !ELEMENT MediaObject (#PCDATA) >

< !ATTLIST MediaObject format (JPEG | MPEGI | MPEGII | Bitmap | wav ...) 'MPEGI' >

< !ELEMENT Delay (#PCDATA) >

< !ELEMENT Jitter (#PCDATA) >

< !ELEMENT LossRate (#PCDATA) >

< !ELEMENT LinkList (Link+) >

< !ELEMENT Link (Start,End, Throughput?, Delay?, Jitter?, LossRate?) >

< !ATTLIST Link type (FixedLink | MobileHostLink | MobileUserLink) 'FixedLink' >

< !ELEMENT Start (#PCDATA) >

< !ELEMENT End (#PCDATA) >

< !ELEMENT Throughput (Average, Burstiness) >

< !ELEMENT Average (#PCDATA) >

< !ATTLIST Average unit CDATA #REQUIRED >

< !ELEMENT AdaptationRule (Condition+, Action+, Notification?, Feedback?) >

< !ELEMENT Condition (#PCDATA) >

< !ATTLIST Average type CDATA #REQUIRED >

< !ELEMENT Action (Component, Method) >

< !ELEMENT Notification (#PCDATA) >

< !ELEMENT Feedback (#PCDATA) >

< !ELEMENT ReconfigRule (Condition+, ReconfigAction, Notification?, Feedback?) >

< !ELEMENT ReconfigAction (#PCDATA) >

< !ELEMENT CPU (Average, Deviation) >

< !ELEMENT ThresholdList (VeryHigh?, High?, Average?, Low?, VeryLow?) >

< !ELEMENT VeryHigh (LowerBound) >

< !ATTLIST VeryHigh type (Bandwidth | CPU | Memory | Disk | Power) 'Bandwidth' >