QoS Specification Languages for Distributed Multimedia Applications: A Survey and Taxonomy

Jingwen Jin Klara Nahrstedt
Dept. of Computer Science
University of Illinois at Urbana-Champaign
{jjin1, klara}@cs.uiuc.edu

***Abstract*—Quality of Service (QoS) is becoming an integral part of currently ubiquitous distributed multimedia applications. However, before any QoS-related mechanisms and policies, such as admission control, resource reservation, enforcement, and adaptation, can be invoked, applications need to express their QoS requirements. A considerable amount of research has been done in QoS-aware Application Programming Interface (API) design and QoS specification language development for multimedia systems. In this paper, we present an extensive survey of existing QoS specification languages, and a taxonomy to classify and compare them. The paper provides readers with a global and insightful knowledge of this important area; knowing how to evaluate QoS languages, and what aspects are most relevant when designing new languages.**

I. Introduction

As the overall computer and communication technology evolves, distributed multimedia applications are becoming ubiquitous, and Quality of Service (QoS) is becoming an integral part of them. Being highly resource (e.g., CPU, memory, and bandwidth) consuming, multimedia applications need resource management at different layers of the communications protocol stack to ensure end-to-end service quality, and to regulate resource contention for fair sharing of resources. However, before any mechanisms and policies of a QoS-aware resource management can be invoked, applications need to specify their QoS requirements and the corresponding resource allocations. Furthermore, they need to describe how quality of service should be scaled and adapted in cases of resource contention or resource scarcity during runtime.

QoS involves a multitude of properties beyond the application-specific aspects, including performance characteristics, availability, responsiveness, dependability, security and adaptivity. In general, QoS specifications (1) should allow for descriptions of quantitative QoS parameters (e.g., jitter, delay, bandwidth) and qualitative QoS parameters (e.g., CPU scheduling policy, error recovery mechanism), as well as adaptation rules; (2) must be declarative in nature, that is, to specify only what is required, but not how the requirement should be carried out; (3) need to be accompanied by a compilation process that maps the QoS specification to underlying system mechanisms and policies.

The main purpose of this paper is to systematically classify and compare the existing QoS specification languages that span across several QoS layers with diverse properties. The provided taxonomy and the extensive analysis will allow us to obtain an insightful view of the existing QoS specification languages along with their properties and relations.

The remainder of this paper is organized as follows. We first describe the taxonomy we use to classify and evaluate existing languages in Section II. We proceed in Sections III, IV, and V by presenting QoS specifications in three layers with representative languages. Evaluation of the languages is presented throughout the text. Mapping between the QoS specifications and underlying system resources becomes a natural topic of discussion when we deal with layered QoS. However, due to space limitations, this subject will only be presented briefly in Section VI. Section VII provides some general comments and concluding remarks.

II. Taxonomy and Evaluation Criteria

Given the heterogeneities of the applications, user preferences, underlying operating systems, networks, and devices, and given the dynamics of their resource usages, it is a complex task, but of great importance, to properly specify QoS requirements and adaptation policies for multimedia processing and delivery. With a large number of QoS specification languages emerging from practical needs, there is a strong need to develop a taxonomy to classify them.

A. Taxonomy

Traditionally, QoS has been a topic discussed mainly in the network communications area. However, for multimedia systems, QoS must be assured not only at the network layer, but at the end systems (e.g., OS and devices) as well [1]. Based on this, we will partition QoS specification languages first into layers according to where in the end-to-end architecture they belong to, and then into classes based on their properties.

1) Layer Partitioning: Layer partitioning is necessary as QoS specifications at different layers present very different features of services. We consider three layers: *user-layer*, *application-layer*, and *resource-layer* (OS and networks), because QoS-rich distributed multimedia systems deploy QoS specifications in these three layers.

1) *user-layer:* At the beginning, a genuine user may need to specify the quality he/she expects to receive from an active application at very abstract level, hence we need *user-layer QoS specification*.
2) *application-layer:* Later on, the human-perceptive quality will be translated into more concrete QoS parameters,

which we call *application-layer QoS specification*. This first mapping between user and application QoS specifications should assume no knowledge of the underlying operating system and network conditions.

3) *resource-layer:* Finally, in order for the application to be executed in a real OS platform and physical network, those application-specific QoS parameters need to be further interpreted into more concrete resource requirements, such as bandwidth/memory allocation and CPU scheduling policies. The resource management requires *resource-layer QoS specifications* to provide the QoS awareness.

Note that an application-layer QoS specification is application-specific, but hardware and platform-independent, while a resource-layer QoS specification depends heavily on the physical world, and is thus hardware and platform-dependent. Application- and resource-layer QoS are specified both quantitatively and qualitatively; the former is used in a specific service as threshold parameters to classify in which quality range the service should operate, while the latter is used to coordinate the involved service as well as to indicate transitions from one operational mode of the service to another.

As a simple example to illustrate the three-layer specification model, we consider a Video-on-Demand (VOD) application where a user wants to see a video located at a remote server. The user may specify, usually through a Graphical User Interface (GUI), the desired overall media quality (at an affordable price) for the VOD service among a set of choices, e.g., high, average and low quality, based on human perception. This specification is then passed to the application layer to derive more application-specific media quality specifications such as video frame rate, image/audio resolution (quantitative parameters), and inter/intra-stream synchronization schemes (qualitative parameter). Translation of these descriptions into physical or logical resource specifications for OS and communication services will take place at next mapping, and the resulting quantitative descriptions at the resource layer may include throughput, delay, delay-jitter, buffer size, and synchronization skew, and qualitative descriptions may include CPU scheduling policies and transmission error recovery mechanisms. A summary of QoS layers and their corresponding QoS issues is given in Table I.

2) Class Partitioning: QoS specification languages can be further classified into classes according to their properties. User-layer QoS specification will only be briefly presented in this paper because, compared to QoS specifications at other layers, it has not been extensively investigated in the multimedia system area. We categorize application-layer QoS specification languages, according to their paradigms, into script-based, parameter-based, process-oriented, logic, markup-based, aspect-oriented, and object-oriented paradigms, and categorize resource-layer QoS specification languages, based on their granularity, into fine and coarse granularity classes.

B. Evaluation Criteria

Since QoS specifications at different layers present very different features of services, we evaluate each layer with different sets of criteria.

User Layer: A genuine multimedia application user is not expected to be a computer expert, thus it is desirable to provide the user with a user-friendly GUI that is simple yet expressive enough as to allow him/her to choose the most appropriate quality within a desired price range. By *expressiveness*, we expect a GUI to provide choices for both quality and pricing. Although there is active research on graphical interface development/evaluation in the Human Computer Interaction (HCI) area largely focusing on aesthetic- or facility-based issues, this paper concentrates on system-related QoS issues only.

Application Layer: Among the three end-system layers, application-layer QoS specifications have been investigated the most, thus will be accordingly devoted a larger space in this paper. We evaluate languages at this layer according to the following five criteria which we deal appropriate:

- *Expressiveness*: A good QoS specification needs to be capable of specifying a wide variety of services, their required resources and corresponding adaptation rules.
- *Declarativity:* A QoS specification should be declarative in nature, so that applications are relieved of the burden of coping with complex resource management mechanisms needed for ensuring QoS guarantees.
- *Independability:* Specifications should be developed independently from the functional code for readability and ease of development/maintenance purposes. Independability also allows a single application to be associated with different QoS specifications at users' request.
- *Extensibility:* This criterion evaluates how easily a language can be extended for specifying new QoS dimensions.
- *Reusability:* Reusability becomes important when QoS specifications get large, as sometimes a new specification may just be an existing one with some minor refinements. A language with reusable features would be favorable in practice. One necessary, but not sufficient, condition for achieving reusability is independability, as the separation would make both functional and non-functional code clean, easy to develop, and easy to maintain.

Resource Layer: At the resource layer, descriptions about the exact physical resource requirement (amount of resource as well as timing of allocation) and adaptation policies are expected. We evaluate resource-layer QoS languages using the expressiveness criterion.

III. User-Layer QoS Specification

A. Characteristics

Multimedia users need to have access and capability to control and customize the quality of their applications. One common way is to provide users with a GUI that is simple, because users are not expected to give sophisticated descriptions about QoS requirements. Therefore, having a user-friendly GUI with a limited number of options that concentrate on subjective user-relevant quality criteria is desirable. There are two main features that user-layer QoS specifications should satisfy: (1) provision of perceptive media quality descriptions (e.g., excellent, good, fair or bad quality) and other related specifications, such

QoS Layers	QoS Issues/Parameters
User Layer (subjective criteria)	• perceptive media quality (e.g., excellent, good, fair, bad) • window size (e.g., big, medium, small) • pricing model (e.g., flat rate, per transmitted byte charge) • range of price (e.g., high, medium, low)
Application Layer (hardware- and platform independent)	• quantitative issues (e.g., video frame rate, image/audio resolution) • qualitative issues (e.g., inter/intra-stream synchronization schemes) • adaptation rules (e.g., if video quality is good, then drop all B frames)
Resource Layer (hardware- and platform-dependent)	• quantitative issues (e.g., throughput, delay, delay jitter, memory size, timing of resource requirements) • qualitative issues (e.g., OS scheduling, reservation style, loss detection/recovery mechanisms) • adaptation rules (e.g., if 80ms $<$ skew $<$ 160ms then drop x video frames)

TABLE I
SUMMARY OF QOS ISSUES FOR THE THREE LAYERS: USER LAYER, APPLICATION LAYER, AND RESOURCE LAYER.

as window size (e.g., big, small), response time (e.g., interactive, batch), and security (e.g., high, low); (2) provision of service pricing choices, i.e., users should be able to specify the range of price they are willing to pay for the desired service. Cost of service is an important factor, because if there is no notion of cost involved in QoS specification, there is no reason for the user to select anything other than the highest level and quality of service [5].

B. *Case Studies*

The INDEX QoS architecture [2] captures, via an intelligent agent (embedded into the general QoS architecture [3]), user's preferences in terms of service quality and price, and maps them into corresponding QoS of Internet network services so that the user's cost-performance relation is optimized. INDEX provides pricing and media quality descriptions via four GUI panels, which reflect billing information, usage information, user preference (price selection and user feedback), and user complaints. The specified QoS information is then used by the underlying agent communication module to select an appropriate Internet service provider to carry out the data communication.

In the QoStalk framework [4], the visual tool allows application programmers to specify, using a hierarchical approach, application service components, where each component is labeled with corresponding application-layer QoS descriptions. Furthermore, application developers provide user-application templates, which include user-application QoS descriptions and their mapping to the corresponding application-layer QoS. A template allows to express media-quality descriptions and non-media-quality descriptions in a simple way, and allows for specifications of different application domains.

The two visual tools have different emphases: the INDEX project concentrates on pricing issues, while the QoStalk project concentrates on descriptions of general QoS issues for multimedia applications. The combination of the two would result in a more expressive user-layer GUI.

IV. APPLICATION-LAYER QOS SPECIFICATION

Two types of application-layer features can be defined: one is *performance-specific*, expressed via quantitative parameters (e.g., frame rate, frame resolution, synchronization skew, level of security), and the other is *behavior-specific*, expressed via qualitative parameters (e.g., what to act if network bandwidth is scarce). A specification language should provide certain abstractions so that programmers do not have to engage themselves into every low-level detail about how/what specific resources and actions need to be invoked. There are two ways to provide programming abstractions: one is through APIs, the other is through language constructs, by either extending existing languages or creating completely new ones. We classify application-layer QoS specification languages based on their paradigms.

A. *Script-Based Paradigm*

Script languages are more abstract than imperative languages, thus are appropriate for specifying things in high level. Roscoe and Bowen [5] present a technique that adds QoS support into Windows NT applications without modifying the applications themselves or underlying operating systems. Their solution is to add a *protocol agent* into the Winsock protocol stack, which intercepts calls made by the application to the networking facilities provided by the underlying operating system, and then contacts the policy daemon containing QoS scripts expressed in SafeTcl [6] (an extension of Tcl with added security issues), to learn how to mark the network packets. Two sample policies, written in SafeTcl, are shown in Figure 1. The first example assigns a higher importance, thus a higher Type of Service (TOS), to network connections initiated by an instance of NetMeeting, and the second example assigns a higher TOS to a particular machine, whose IP address is 199.2.53.102.

The approach was mainly to allow existing applications to take advantage of QoS facilities described by the DiffServ framework[1]. Hence, SafeTcl scripts were only used to instruct

[1]DiffServ is a proposed IETF model for offering differentiated services in the Internet, by marking IP packets with a byte value known as the DS field specifying how the packet should be treated on a per-hop basis by routers. More details will be given in Section V-B.2.

```
# Sample 1: Give NetMeeting better network service
...
proc Socket {pid uid cmd family type protocol fd}{
    if{[string compare $cmd $NetMeetingStr] == 0}{
      if{$family == $AF(INET) && $type == $SockType(DGRAM)}
        {return "TOS 48"}
    }
    else {return "TOS 16"}
  }
}

# Sample 2: Give different service to a particular destination
proc Connect {pid uid cmd fd address sqos gqos}{
    ...
    if {$inaddr == "199.2.53.102"}{return "TOS 48"}
}
```

Fig. 1. Two examples of QoS scripts written in SafeTcl.

the protocol agent in marking packets (although the language itself is far more expressive). It remains to be seen how well the language can be extended to specify other QoS dimensions. This approach allows specifications to be written syntactically separate from the applications, thus has good independability.

B. Parameter-Based Paradigm

In this approach, which is widely adopted, application developers define data structures to express qualitative and quantitative parameters, without creating a new QoS language; it relies on the underlying QoS management architecture to evaluate and act on the parameters.

An example of the parameter-based paradigm is QoS-A, developed by Campbell [7], which uses a Quality of Service Architecture (QoS-A) to deal with QoS enforcement both at end systems and in the networks in a uniform way. A QoS parameter specification between two communicating parties defines a service contract that includes several aspects: flow specification, QoS commitment, QoS adaptation, QoS maintenance, reservation style, and cost. The service contract is implemented as a C structure, and each clause again is represented as another structure.

The *flow specification* structure specifies performance-related quantitative metrics, such as frame size, frame rate, burst size, peak rate, delay, jitter, and loss rate[2]. The *QoS commitment* clause describes the requirements in a qualitative way, e.g., whether the services are guaranteed, statistical, or best-effort. The QoS *adaptation* structure is used to specify which remedial actions to take in the presence of QoS violation, where adaptation actions can be triggered upon the degradation of, for example, loss, jitter, throughput, and delay. An action can be like "if the maximum end-to-end delay is exceeded then the QoS-A will inform the user of the QoS event via an upcall". The QoS *maintenance* structure provides choices for an application to specify how frequent or how well it wants to be notified of performance changes. The *reservation styles* structure allows to specify resource reservation styles, e.g., *fast*, *negotiated*, and *forward*. Lastly, the *cost* structure allows applications to specify the range of price or payment mode that the user is willing to follow.

[2]Note that a fbw specifi cation actually spans across application and resource layers.

The set of QoS constructs is rather rich, making the API good in terms of expressiveness, and we believe new QoS constructs can be added without too much difficulty. QoS-A also allows contracts to be developed independently of the applications. Since QoS parameters and actions are specified as structures, QoS-A does not provide special facilities for specification reuse.

C. Process-Oriented Paradigm

In the process-oriented paradigm, processes, as units of execution, communicate and synchronize with one another through message passing, or communication ports. Process-oriented QoS specification allows to associate QoS with communicating end ports, as well as to express negotiation of QoS constraints and monitoring of QoS between two ports.

An example of process-based QoS specification language is QuAL (Quality-of-service Assurance Language), developed by Florissi [8]. QuAL is extended from Concert/C, and its language constructs provide means for specification and negotiation of QoS constraints, specification of QoS violation handlers, and customization of QoS monitoring. QuAL supports handling of two types of QoS metrics: application-specific QoS metrics and resource-specific QoS metrics. This section concentrates on the former. Resource-specific QoS metrics of QuAL will be presented in Section V-B.2.

At application layer, metrics such as frame rate and synchronization are of interest. QuAL monitors these metrics through function calls. For example, the command *qual_monitor* can be invoked if an application wants to monitor the inter-arrival delay of the incoming video frame at a certain port. QuAL allows specification of filters, which are inspectors placed in ports to check the data flow to guarantee that only complying messages are injected into the communication stream. This feature may be useful if receivers are heterogeneous, because the sender may specify several filters such as $low\_quality$, $med\_quality$, and $high\_quality$, and let receivers tune into one of them according to their own capacity. Some other important features of QuAL include automatic QoS violation monitoring by having the application inform the QuAL runtime which conditions identify a QoS violation, so that when a violation is detected, the runtime notifies the application of the occurrence.

QuAL has a good expressiveness, and one should be able to easily develop new QoS constructs when needed. However, specifications written in QuAL are spread across the functional code, making both parts hard to develop and maintain. In addition, the language is more instructive than declarative. Like all previous languages, QuAL does not present features for reuse.

D. Control-Based Logic Approach

Control-based approaches are adopted in adaptive systems for QoS specifications of adaptive policies and flow control. Some systems use adaptive control techniques such as PID (Proportional-Integral-Derivative) controller to specify and control fine granularity when scheduling flows or tasks. Other systems use fuzzy-control techniques to specify and assist in adaptation of QoS [9].

The fuzzy-control QoS specification [9] is fully supported by an underlying middleware control architecture. This architecture enforces the adaptive application to behave according to the fuzzy-control specification, and it comprises two components: the adaptor and the configurator. The adaptor makes control decisions with global awareness of application QoS requirements and resource availability of the entire system. The configurator uses the fuzzy control QoS specification and translates the normalized control decisions, generated by the adaptor, into actual parameter-tuning actions to be used during the execution of the application.

The fuzzy control approach allows applications to express QoS-aware adaptation policies and preferences in forms of rules and member functions. Rules are specified in an *if-then* format: "if X_1 is A_1 and ... and X_n is A_n then Y is B", where $X_1, \ldots, X_n$ and Y are parameters corresponding to certain QoS-relevant system conditions such as bandwidth and CPU, and $A_1, \ldots, A_n$ and B represent actual values of parameters in the fuzzy form *high*, *low*, *moderate* or *below_average* for the linguistic variable *cpu_demand*. An example of QoS-aware adaptation rule is "if *cpu_availability* is *very_high* and *available_bandwidth* is *very_low* then *rate_demand* is *compress*", which tells the system to compress the data because available bandwidth is very low but there is large amount of cpu available.

One limitation of the fuzzy-control approach is that it is intended to specify only actions, but not other QoS properties. Thus this approach is poor in terms of expressiveness and is not sufficient for multimedia applications in general.

E. Markup-Based Paradigm

Extensible Markup Language (XML) [10] 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. A markup language is a mechanism to identify structures in a document, and the XML specification defines a standard way to add markups to documents. Different from HTML, where both the tag semantics and the tag set are fixed, XML specifies neither semantics nor a tag set. In fact XML is really a meta-language for describing markup languages; i.e., it provides a facility to define tags and the structural relationships between them. Since there is no predefined tag set, there cannot be any preconceived semantics. All of the semantics of an XML document will be defined either by the applications that process them or by stylesheets[10].

HQML in [4] is a QoS language based on the XML standards, which basically defines a set of tags relevant to QoS of multimedia applications, to allow application developers to define QoS parameters and policies. A sample HQML specification, whose *id* is 1, is depicted in Figure 2. Tags such as <ServerCluster>, <ClientCluster>, and <LinkList> are used to define the QoS requirements of the server, gateway, client machines, and the properties of the links (e.g., fixed link or mobile link). HQML allows to specify adaptation rules between the pair of tags <ReconfigRuleList> and </ReconfigRuleList>. For example, in Figure 2, the adaptation rule indicates that "when Bandwidth is very low, then the application execution should switch to specification whose id is 2". A good feature of HQML is that it can, due to XML's meta-language property, be easily extended to include new QoS parameters. The language is also good in expressiveness, declarativity, and independability. However, it does not have any special constructs that facilitates the extension and reuse of existing specifications.

```
<AppConfig id = "1">
  <ServerCluster>
      ...
  </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>
      ...
  </LinkList>
  <ReconfigRuleList>
    <ReconfigRule>
      <Condition type = "Bandwidth"> very low </Condition>
      <ReconfigAction type = "switch to"> 2 </ReconfigAction>
    </ReconfigRule>
  </ReconfigRuleList>
</AppConfig>
```

Fig. 2. Sample HQML specification.

F. Aspect-Oriented Approach

Many distributed systems are built on top of CORBA - a middleware that provides a flexible communication and activation substrate for distributed heterogeneous object-oriented computing. CORBA hides system- and network-specific characteristics of objects behind the standardized Interface Description Language (IDL) specifications, so that the objects exhibit only their functional interfaces. The fact that IDL abstracts away low-level details simplifies development and maintenance of distributed objects, but at the same time, it also makes the inclusion of non-functional features (such as QoS) into the system difficult because much of the information required to support the QoS also becomes hidden.

The Object Management Group (OMG) has put some effort on extending CORBA to support QoS-enabled applications[11]. However, so far there does not seem to be any concrete specification language developed at OMG. CORBA IDL has been extended by Becker and Gheis [12] with constructs for QoS characterizations. However, this approach statically binds QoS characterizations to interface definitions[3], therefore does not allow different properties to be associated with different implementations of the same functional interface. In the

[3]This is an approach similar to that adopted by TINA ODL[13], which also syntactically includes QoS requirements within interface definitions. This way, each functional interface can be only associated with fixed QoS properties.

research community, two different language approaches for distributed object-base applications have been developed: *aspect-oriented approach* and *object-oriented approach*. This section concentrates on the Aspect-Oriented approach, Section IV-G will present Object-Oriented approach.

The Aspect-Oriented approach follows the Aspect-Oriented Programming (AOP) paradigm developed by Kiczales et al [14], because QoS-related tasks are examples of the so-called *aspects* in this paradigm. Aspects are not units of the system's functional decomposition; rather, they are properties that affect the performance or semantics of the components in systematic ways. Using traditional programming languages, implementations of such aspects would result in tangled code with aspect-related code spreading over the program and cross-cutting the basic functional components of the system. The Aspect-Oriented Programming technology has been developed to support clean abstraction and composition of both aspects and functional components. Using AOP, an application can be decomposed into functional components and aspects, where different aspects can be programmed in different languages suitable to the tasks, and at the end a special language processor, called *aspect weaver*, would coordinate the co-composition by weaving all the code together to produce a single executable application.

The Aspect-Oriented approach can be found in the Quality Object (QuO) framework [15], [16] developed at BBN. QuO 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, and extends the CORBA functional IDL with a QoS Description Language (QDL) consisting of three sub-languages, the *Contract Description Language* (CDL), the *Structure Description Language* (SDL), and the *Resource Description Language* (RDL).

CDL is used for specifying a QoS contract, which consists of four major components: a set of *nested regions*, each representing a possible state of QoS; *transitions* for each level of regions specifying behavior to trigger when the active region changes; references to *system condition objects* that gather runtime information for measuring and controlling QoS; and *callbacks* for notifying the client or object. The last two components are specified as contract parameters, and are usually shared among several contract instances. The nested regions, describing the relevant possible states of QoS in the system, are defined by predicates on the values of system condition objects. The regions are evaluated by the contract to determine whether they are active. If the currently active regions have suffered changes since the last contract evaluation, transition behaviors are triggered. A QoS contract written in CDL is given in Figure 3. The example shows a client contract that can operate in two possible operating modes: *Low_Cost* and *Available*. *ClientExpectedReplicas* and *MeasuredNumberReplicas* are system condition objects indicating the client's expected number of replicas and the actual number of replicas available, respectively. The reality region transitions are used to notify the client about changes in the number of replicas. For example, in the *Low_Cost* mode, if *MeasuredNumberReplicas* drops from *High* to *Low*, then a callback *availability_degraded()* will be triggered. The negotiated region transitions are used to specify actions to take when

```
contract repl_contract(
 ...
 negotiated regions are
  region Low_Cost : when ClientExpectedReplicas == 1 =>
   reality regions are
    region Low    : when MeasuredNumberReplicas < 1 =>
    region High   : when MeasuredNumberReplicas >= 1 =>
    transitions are
     transition High->Low : ClientCallback.availability_degraded();
    end transitions;
   end reality regions;
  region Available : when ClientExpectedReplicas >= 2 =>
   reality regions are
     ...
    transitions are
     ...
    end transitions;
   end reality regions;
  transitions are
    transition Low_Cost->Available :
        ReplMgr.adjust_degree_of_replication(ClientExpectedReplicas);
    transition Available->Low_Cost :
        ReplMgr.adjust_degree_of_replication(ClientExpectedReplicas);
  end transitions;
 end negotiated regions;
end repl_contract;
```

Fig. 3. Sample CDL contract.

the client changes its desired replication.

While CDL is used for describing the QoS contract between a client and an object, SDL allows programmers to specify the structural aspects of the QoS application. This includes adaptation alternatives and strategies based on the QoS measured in the system. The current version of SDL enables us to express behaviors to invoke for method calls/returns based on the current regions of contracts when the calls/returns occur. Such behaviors can have descriptions like: "when the client desires higher availability than what is measured, i.e., the contract is in the *Available* negotiated region and the *Low* reality region, then an exception will be thrown". The QuO code generator takes the normal CORBA IDL code, as well as specifications written in SDL and CDL[4], as the input, and weaves them into a single application.

As we can see, QDL is good in most of our evaluation criteria; it is expressive, declarative, independent, and extensible. However, like the previous languages, there is no special facility for specification reuse.

G. Object-Oriented Approach

Until this point, the languages described in this paper did not take reusability into their designs. We borrow the terminology "Object-Oriented" from the traditional languages area to concentrate on the specification refinement issue that, much like the class inheritance concept in Object-Oriented languages, helps to enhance specification reusability. Many language developers may have thought that specifications, unlike the traditional functional code, should be small and relatively unrelated. However, as the QoS-awareness in multimedia applications increases, we may expect specifications of larger sizes that are closely related to each other based on their properties.

[4] Although QDL was planned to be a suite of three different languages, the third sub-language, RDL, which was supposed to abstract the physical resources used by the object, never really came out.

```
type Reliability = contract {
  NumberOfFailure: decreasing numeric no/year;
  TTR: decreasing numeric sec;
  Availability: increasing numeric;
};
type Performance = contract {
  delay: decreasing numeric msec;
  throughput: inceasing numeric mb/sec;
};

systemReliability = Reliability contract {
  numberOfFailures < 10 no/year;
  TTR {
       Percentile 100 < 2000;
       Mean < 500;
       Variance < 0.3
  };
  availability > 0.8;
};

ServerProfile for ServiceInterface = profile {
  require systemReliability;
  from operation1 require Performance contract {
       delay {
         percentile 50 < 10 msec;
         percentile 80 < 20 msec;
       };
  };
  from operation2 require Performance contract {
       delay < 4000 msec
  };
}
```

Fig. 4. Example of contracts and profi le written in QML.

A very good example that provides specification refinement features is QML (QoS Modeling Language) [17], developed at the HP Laboratories. QML is for Corba-based distributed object systems, and offers three main abstraction mechanisms for QoS specification: contract type, contract, and profile; where contract type defines the dimensions that can be used to characterize a particular QoS aspect, contracts are instances of contract types, and profiles associate contracts with interfaces and operations. These concepts are illustrated in Figure 4. *Reliability* and *Performance* are two contract types each with its own dimensions, e.g., number of failures per year, time to repair a failed service, availability of the system. The contract *systemReliability* is an instance of contract type *Reliability* with constraints associating with the dimensions defined in the contract type. Lastly, the profile *ServerProfile* is defined for an IDL service interface (named *ServiceInterface*, with two operations: *operation1* and *operation2*). The profile requires the two previously defined contracts either for the service in general (*systemReliability* - which should hold for all operations) or for certain specific operations.

The QoS refinement features in QML are actually consequences of the facts that class inheritance allows an interface to be defined as a refinement of another interface and that QoS specifications are associated with interfaces (via profiles). Two kinds of refinement, contract refinement and profile refinement, are supported in QML. A contract B refined from a contract A is specified as *B = A refined by* $\{\dots\}$ where A is the base contract, and the contract enclosed in the curly brackets is a delta that describes the differences between the contracts A and B by specifying either those QoS properties omitted in A or replacing specifications in A with stronger ones. Profiles can be refined in a similar way, with the delta specifying new contract association to be added or strengthened in the new profile.

Among all application-layer languages presented so far, QML supports specification reusability the best through contract and profile refinement. It is also good in terms of independability and extensibility. However, one limitation of QML is that for each contract type (e.g., reliability) that a profile involves, at most one contract can be used as a default contract within the profile. For example, the profile *ServerProfile* declares that it requires *systemReliability*, which is an instance of the contract type *Reliability*, it cannot require a second instance of the same contract type. It would be good if future work could allow a profile to require several contracts of the same type to be the default type and take the final result as, for example, the set of strongest constraints defined in all contracts. QML is a general-purpose QoS specification language capable of dealing with any QoS aspects (e.g., reliability, availability, performance, security, and timing) and any application domain. However, one limitation is that it largely specifies QoS properties at design time, but does not address the problem of what actions to take at runtime if the QoS requirements cannot be satisfied in the current execution environment. In this respect, QDL is more expressive than QML.

H. Comparisons of Application-Layer QoS Specification Languages

To better view a high-level picture of the application-layer QoS specification languages' performances according to our evaluation criteria, we provide a simple table in this section (Table II), where each evaluation criteria is attributed three values: good, fair, and poor, based on our own view.

V. Resource-Layer QoS Specification

Application-layer specifications only state requirements in a rather high-level, abstract way. Later, these requirements are further translated into more concrete resource demands. That is, descriptions such as which physical resources will be needed for the application, when they need to be allocated, which mechanisms should be adopted, and which transport protocol is to be used, need to be provided. We classify specifications at this layer according to their granularity into: coarse granularity and fine granularity categories. By coarse granularity, we only expect a meta-level specification, while by fine granularity, we expect concrete descriptions of required resources.

A. Coarse-Granularity Resource-Layer QoS Specification

1) Characteristics: Some resource-layer QoS specifications only specify resource requirements in a rather abstract way. For example, they may specify what (amount of) resource is required, but do not care about when the resources need to be allocated, or what action to take if the resource requirement can not be met, or if several resource instances (e.g., processors) are available, which specific one to use. We call languages that do not allow descriptions of fine-granularity resource requirements coarse-granularity languages.

QoS Language	expressiveness	declarativity	independability	extensibility	reusability
SafeTcl	poor	fair	good	poor	poor
QoS-A	good	good	good	good	poor
QuAL	good	poor	poor	good	poor
Fuzzy Control	fair	good	good	fair	poor
HQML	good	good	good	good	poor
QDL	good	good	good	good	poor
QML	fair	good	good	good	good

TABLE II
COMPARISON OF APPLICATION-LAYER QOS LANGUAGES.

2) Case Studies: The Resource Specification Language (RSL) is developed by the Globus project [18] and is used to communicate requests for resources between components in metacomputing systems. The authors developed a hierarchical resource management architecture comprising several components, namely, resource broker, resource co-allocator, local resource manager, and an extensible resource specification language - RSL.

Initially, an application specifies its QoS requirement in RSL. This specification is of high-level in that the required items may be physically distributed in several locations and systems. This high-level specification is passed through resource brokers that can translate it into more concrete resource requirements and locate required resources. This translation generates a specification, called a ground request, in which the locations of the required resources are completely specified. A co-allocator is then responsible for coordinating the allocation and management of resources at multiple sites, by breaking the multi-request (involving resources at multiple sites) into several requests and passing them to the appropriate local resource managers. The syntax of RSL is very simple. An RSL specification is constructed by combining simple parameter specifications and conditions with logic operators &, $|$, and +. As an example, the multi-request below shows that the executable program, $myprog$, requests 5 nodes with at least 64 MB memory, or 10 nodes with at least 32 MB memory.

```
&(executable=myprog)
(|(&(count=5)(memory>=64))(&count=10)(memory>=32))
```

A ground request that results from the interpretation of brokers would further specify information about which resource manager will be handling particular requirements in the multi-request, so that a co-allocator can determine to which resource manager each component of the multi-request should be submitted.

As we can see, RSL does not deal with timing of resource allocation or resource scaling/adaptation. Since multimedia applications are very sensitive to timing, and are adaptive in a sense that resource requirements are usually flexible (instead of rigid), a coarse-granularity language does not suit well for multimedia applications.

B. Fine-Granularity Resource-Layer QoS specification

1) Characteristics: For multimedia services, specifications of finer granularity are required. We expect from a fine-granularity resource-layer QoS specification descriptions of (1) quantitative and qualitative QoS requirements; (2) timing of the resource requirements, i.e., for when and for how long the resource needs to be allocated; and (3) adaptation rules.

2) Case Studies : Three examples will be presented in this section: DSRT, QuAL, and IntServ/DiffServ.

DSRT: The purpose of DSRT (Dynamic Soft Real Time System), developed by Chu [19], is to support soft real-time applications in traditional time sharing systems by developing a middleware between applications and the operating system. This new layer consists of some APIs developed in C++ that allow applications to reserve and free cpu resources. The APIs also define some structures that allow users to specify the amount of cpu resources required during the application execution (e.g., period, peak processing time, burst tolerance), and some adaptation strategies (e.g., upper and lower bounds on the guaranteed parameter that can be adjusted by the DSRT system). Once these values are set, a function call *cpu.reserve(reservation)* and *cpu.setAdaptStrategy(strategy)* will, respectively, make cpu reservation and set some adaptation strategies according to the values specified in the *reservation* and *strategy* structures. The author also extended the same mechanism from cpu to other resources such as memory and communication.

QuAL: QuAL [8] provides a range of QoS attributes for the specification of network and OS resource-layer QoS metrics. As an example, imagine a situation where an application A periodically sends images to a remote site B. The applications at both sides may specify QoS constraints on the transmission of images as in Figure 5. Having both sites specified their requirements, QuAL abstracts QoS negotiation between peer applications by type checking connecting ports and guarantees that two ports are connected only if they have compatible QoS attributes; i.e., if the compiler and runtime are able to coerce all the QoS requirements of the sender into the QoS requirements of the receiver, or vice versa. In most cases, coercion is possible when the QuAL compiler or runtime can upgrade a less restrictive constant until it matches a more restrictive one. QuAL also allows specification of actions to perform when QoS violations occur. The runtime automatically monitors QoS delivery and invokes application-customized exception handlers when violations are detected. The runtime scrutinizes interactions among applications, communication protocol stacks, and OS, and collects statistics on the delivered QoS into a QoS Management Information Base (QoS MIB), which is then used by applications to dynamically adjust their execution. QuAL does not deal with

```
realtm {loss 6;
    rate sec 10 - sec 30;
    delay ms 40;
    jitter ms 33;
    recovery sec 4;}
receiveport {image_t} *image_output;

realtm {loss NULL;
    rate sec 10 - sec 25;
    jiter ms 33, nocoercion;
    recovery sec 3;}
receiveport {image_t} image_input;
```

Fig. 5. Specifi cation of QuAL QoS constraints on the image transmission at site A & B.

timing of resource requirements, thus in this respect, it is less expressive that DSRT.

IntServ/DiffServ: The vigorous interest in QoS issues within the Internet community has led the rapid development of two IP standards by IETF: the Integrated Service (IntServ) based on the Resource ReSerVation Protocol (RSVP) and the Differentiated Services (DiffServ). With the growing interest in Internet audio and video, IntServ became a standard focusing on per-flow QoS, where admission control is invoked at each node to make a local accept/reject decision by the signaling protocol RSVP. The poor scalability of IntServ (due to the complex signaling protocol and state maintenance overhead incurred at routers) has, later, led to the development of DiffServ which, rather than providing QoS on a per-flow base, considers flow aggregates at the edge of the network to keep the core of the network simple. By marking packets' type field (Type of Service field in IPv4 and Traffic Class in IPv6) at the edges of network, the core of the network only needs to check this type field, which represents a small and well-defined forwarding behaviors, to make forwarding decisions. As can be seen, QoS for IP networks is largely to quantify and enforce the treatment a particular packet can expect as it transits a network.

VI. Discussions

QoS mapping refers to the process of translating higher-level representations of QoS into lower-level representations of QoS. The translation between user QoS and application QoS is nontrivial and is still an open research issue, because the perceptual issues are not completely understood [1], and because human perceptions for quality may change as the technology evolves.

Automated QoS mapping between application- and resource-layer can shield applications from the complexity of underlying QoS specifications and QoS management functions. QoS mapping development between these two layers is still in its infancy. Most of the research to date has focused primarily on deriving appropriate QoS parameters for memory, CPU processing and network connections in a rather static, architecture-specific manner [7]. Individual work such as [20], [21] largely deal only with partial mapping rules. They either provide only quantitative translations of certain parameter value into another (e.g., mapping from picture resolution to bandwidth or memory requirement), or concentrate only on mapping for specific application based on a specific architecture. Although QuAL[8] was intended for specifying both application and resource-layer QoS requirements, it did not deal with issues of mapping between the two layers. The newest and most significant development in QoS mapping can be found in [22].

The difficulty or discouragement related to QoS mapping development may be caused by the fact that the underlying operating systems and networks are not fully prepared to support QoS appropriately yet. For example, it makes no sense to specify advanced resource reservation at a higher level if the lower levels do not enforce such a feature physically. It is probable that general rules for mapping may be impossible to derive, given that there are varieties of operating systems, networks, and applications. Therefore, mapping rules are very likely to be system- and application dependent.

VII. Conclusions

In this paper, we reviewed QoS languages currently existent in the literature, and classified and compared them according to our taxonomy and criteria. As we can see, there are already quite a lot of specification languages for depicting QoS requirements in different situations. There are certainly many other languages left out from this paper due to space limitation. For example, in [23], Staehli et al defined QoS for multimedia database systems by making strong distinctions between contract, view, and quality specification. They used the mathematical notation - Z, to denote the specifications. Since Z specifications are purely declarative and inherently non-executable, and since most of the Z constructs are too abstract to be refined to real implementations automatically by existing translation tools, such specifications are only suitable for helping derive implementations.

There is, as yet, no consensus on the precise set of dimensions that quality of service should encompass. Much of the current effort centers on providing assurances for attributes such as cost, timeliness (e.g. response time, jitter), volume (throughput), precision, accuracy, synchronization, availability, reliability, and security.

Aspects of QoS management can be inserted into applications in a variety of ways. For example, they can be specified at the application layer while the mechanisms and enforcement are provided by the OS and communication systems; they can also be embedded into the resource infrastructure (e.g. communication network), effectively hiding from the application. Since QoS at each layer has different purposes, it will most likely become prevalent in all layers either for the purpose of ease of specification at high layers or for the purpose of resource enforcement at low layers. Actually, the boundary of QoS layers get somewhat blurred in some specification languages presented in this paper. For example, QoS-A and QuAL deal with both hardware-independent and hardware-dependent QoS parameters and adaptations. Low-level, hardware-dependent QoS specifications are usually too complex for application developers to derive. Ideally, this should be tasks of the mapping process. Therefore, future work should give more importance to the understanding and derivation of a comprehensive QoS mapping system in order to alleviate high-level programmers of the burden of learning characteristics of low-level physical resources.

VIII. Acknowledgments

This material is based upon work supported by DARPA (under Award No. F30602-97-2-0121), NSF (under Awards No. CCR-9988199 and EIA 99-72884 EQ), NASA (under Award No. NAG2-1406). The authors would like to thank the anonymous reviewers for their detailed and insightful comments.

References

[1] Klara Nahrstedt and Jonathan M. Smith. The QoS Broker. *IEEE Multimedia Magazine*, 2(1):53–67, 1995.

[2] J. Altmann and P. Varaiya. INDEX Project: User Support for Buying QoS with Regard to User's Preferences. Napa, CA., May 1998. Sixth International Workshop on Quality of Service (IWQoS98).

[3] C. Aurrecoechea, A. T. Campbell, and L. Hauw. A Survey of QoS Architectures. *ACM/Springer Verlag Multimedia Systems Journal, Special Issue on QoS Architecture*, 6(3):138–151, May 1998.

[4] Xiaohui Gu, Klara Nahrstedt, Wanghong Yuan, Duangdao Wichadakul, Dongyan Xu. An XML-based Quality of Service Enabling Language for the Web. *Journal of Visual Language and Computing, special issue on Multimedia Languages for the Web (Academic Press)*, 13(1):61–95, Feb. 2002.

[5] T. Roscoe and G. Bowen. Script-driven Packet Marking for Quality of Service Support in Legacy Applications. In *Proceedings of SPIE Conference on Multimedia Computing and Networking 2000*, San Jose, CA, Jan. 2000.

[6] J. K. Ousterhout, J. Y. Levy, and B. B. Welch. The Safe-Tcl Security Model. Technical Report TR-97-60, Sun Microsystems Laboratories, Mar. 1997.

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

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

[9] Baochun Li, Klara Nahrstedt. Dynamic Reconfiguration for Complex Multimedia Applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, Florence, Italy, Jun. 1999.

[10] Norman Walsh. What is XML? O'Reilly XML.com, Oct. 1998. http://www.xml.com/pub/a/98/10/guide1.html#AEN58.

[11] Object Management Group. Quality of Service. OMG green Paper, draft revision 0.4a edition, Jun. 1997.

[12] Christian R. Becker and Kurt Gheis. Maqs - Management for Adaptive QoS-enabled Services. In *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, Dec. 1997.

[13] Telecommunications Information Networking Consortium. TINA Object Definition Language, Jun. 1995.

[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland Springer-Verlag LNCS 1241, Jun. 1997.

[15] 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*, Apr. 1997.

[16] 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, Springer - Verlag*, 1511, 1998.

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

[18] Ian Foster, Carl Kesselman. The Globus Project: A Status Report. In *Proceedings of IPPS/SPDP'98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[19] Hao-hua Chu. *CPU Service Classes: A Soft Real-Time Framework for Multimedia Applications*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

[20] Jean-Francois Huard and Aurel A. Lazar. On QoS Mapping in Multimedia Networks. In *21th IEEE Annual International Computer Software and Application Conference (COMPSAC'97)*, Washington DC, Aug. 1997.

[21] Kentarou Fukuda, Naoki Wakamiya, Masayuki Murata, and Hideo Miyahara. QoS Mapping between User's Preference and Bandwidth Control for Video Transport. In *Fifth International Workshop on Quality of Service (IWQoS'97)*, New York, NY, May. 1997.

[22] Duangdao Wichadakul. *Q-Compiler: Meta-Data QoS-Aware Programming and Compilation Framework*. PhD thesis, Computer Science Department, University of Illinois at Urbana Champaign, Jan. 2003.

[23] Richard Staehli, Jonathan Walpole and David Maier. Quality of Service Specification for Multimedia Presentations. *Multimedia Systems*, 3(5/6), Nov. 1995.