QoSME: an Environment to Manage Multimedia Application QoS*

Patrícia G. S. Florissi
Departamento de Informática
Univ. Federal de Pernambuco
PO Box 7851, Recife, PE 50732-970
Brazil
pgsf@di.ufpe.br

and

Yechiam Yemini
DCC Lab
Columbia University
450 Computer Science Bldg., NYC, NY
10027 USA
yy@cs.columbia.edu

Resumo

Este artigo introduz o Ambiente de Gerenciamento de Qualidade de Servico (QoSME) que permite o gerenciamento (ou seja, a negociação, o monitoramento, a adaptação e o controle), por parte de aplicações, da Qualidade de Serviço (QoS) oferecida pela rede. Aplicações em QoSME acessam protocolos de transporte da rede via QoSockets, uma nova extensão do mecanismo de sockets que inclui ferramentas para negociação de QoS por parte de aplicações. QoSockets mapeia restrições de QoS especificadas pelas aplicações em serviços (de QoS) específicos do ambiente de suporte e, consequentemente, encoberta a distância entre a QoS especificada por aplicações e aquela oferecida pela rede. QoSockets esconde a heterogeneidade nas interfaces a nível de transporte e abaixo e facilita a portabilidade e reuso, simplificando o desenvolvimento e manutenção de aplicações que requerem QoS. QoSockets automatcamente monitora a QoS obtida e armazena em Bases de Informação de Gerenciamento (MIBs) rastreamentos da QoS realmente obtida por cada aplicação. Quando há violação de QoS, QoSME invoca mecanismos de tratamento de exceções que executam processos definidos pelas aplicações para atuar sobre os eventos errôneos, promovendo adaptação costumizada por aplicação à QoS obtida. Agentes de QoS MIBs do Simple Network Management Protocol (SNMP), embutidos em QoSME, integram gerenciamento de QoS a nível da aplicação com sistemas tradicionais de gerenciamento de redes. Eles provêm acesso a QoS MIB para administradores de redes que monitoram ambos, as necessidades de aplicações e a QoS realmente obtida pela conexão. Além do mais, os administradores podem adaptar a forma de gerenciar seus recursos para melhor casar as necessidades das aplicações com a QoS realmente obtida.

Abstract

This paper introduces a novel Quality of Service Management Environment (QoSME) that enables applications to manage (i.e., negotiate, monitor, adapt, and control) QoS delivery by a network. QoSME applications access network transport protocols via QoSockets, a new extension of the sockets mechanism that includes capabilities for QoS negotiation by applications. QoSockets maps application specified QoS constraints into underlying environment specific QoS requests and bridges the gap between the QoS requested by applications and the QoS supported by the underlying network. QoSockets shelters heterogeneity at the transport level and below and eases their portability and reuse, simplifying the development and maintenance of QoS demanding applications. QoSME automatically monitors QoS delivery and gathers into QoS Management Information Bases (MIBs) traces on QoS performance per application. Upon QoS violations, QoSME signals application-provided exception handlers to act upon faulty events, promoting application customized adaptation to QoS delivery. QoS MIB Simple Network Management Protocol (SNMP) agents embedded in QoSME integrates application level QoS management into standard network management frameworks. They provide QoS MIB access to network managers that monitor both application needs and end-end QoS delivery. Furthermore, managers adapt their resource management to improve overall match between end-end QoS delivery and application needs.

* This work was performed while the first author was pursuing her Ph.D. in Computer Science in the Distributed Computing and Communications (DCC) Lab at Columbia University.

1 Introduction

End users are sensitive to the performance of multimedia communications. For example, excessive jitter in a voice stream results in illegible speech. Similarly, significant delays in delivering interactive video conference renders cumbersome interactions. Jitter, delay, and throughput are but a few of the *Quality of Service* (*QoS*) metrics measuring the performance of communications. In general, distributed isochronous applications are sensitive to various QoS metrics.

It is therefore necessary to assure that the QoS expected by applications and the QoS delivered by the network match closely. This gives rise to the following challenges:

1. *How should a network interact with applications to assure end-end QoS?* Several answers have been proposed. One possibility is for applications to request the network explicitly the QoS that they desire and for the network to adapt to these requests. For example, a connection-based application could specify at connection establishment time the maximum traffic rate that it requires. The network will adapt its bandwidth allocation accordingly. At another extreme, the network would provide optional end-end channels with assured QoS. The application will select the channel that best matches its QoS needs. Another possibility is for the network to monitor the behavior of applications and adapt its delivery to their traffic patterns.

2. *How should the network configure its resources to assure end-end QoS delivery?* End-end QoS delivery is the sum of traffic handling and resource allocation at intermediate nodes. The network must thus map end-end QoS needs into respective configuration and handling at intermediate nodes. This is complicated by the fact that end-end paths may be stretched through multiple domains using heterogeneous intermediate nodes components.

3. *How can applications adapt to the actual QoS delivered by the network?* For applications to adapt, they must be able to monitor QoS delivery and to detect failures in meeting their needs. They must be able to reconfigure their computations to adapt to variations in the QoS delivery. They must support these functions independently of the underlying transport and network mechanisms.

The first and second questions have been considered by various publications [braden95, hyman93, lazar90, topolcic90]. These works have focused primarily on mechanisms at the transport layer and below to support QoS delivery. In contrast, this paper introduces applications-layer technologies to address these challenges and focuses primarily on the third challenge.

This paper introduces a novel *QoS Management Environment* (*QoSME*) to support effective adaptation and control by applications of QoS delivery. QoSME enables applications to monitor, analyze and adapt to the QoS delivery by the network. It also provides mechanisms that permit applications to convey their QoS needs to their end-node system. This permits end-node mechanisms to control the QoS delivered by the network to meet application needs.

QoSME also enables novel solutions to the first and second challenges discussed above. The network can monitor both application needs and end-end QoS delivery. It can use this data to adapt its resource management to improve the overall match between end-end QoS delivery and applications needs.

QoSME fully automates QoS monitoring. The instrumentation to monitor QoS is generated automatically as a side effect of application access to network transport via *QoSockets*, an extended socket mechanism provided by QoSME. QoSME monitors QoS constraints set by applications through QoSockets calls. It detects violations of these constraints and invokes automatically respective handlers provided by applications. This enables applications to adapt to the actual QoS delivered by the network.

QoSME integrates QoS management within standard network management frameworks. The data collected by QoSME instrumentation is organized in QoS *Management Information Bases* (*MIBs*) accessible via the *Simple Network Management Protocol* (*SNMP*)[1]. This permits external managers to monitor end-end QoS delivery and adapt network resource allocation and operations accordingly.

This paper is organized as follows. Section 2 discusses the QoSME architecture. Section 3 analyzes application interactions with network QoS delivery mechanisms. Section 4 addresses QoS negotiation between peer applications. Section 5 discusses how applications can program QoS metrics to be monitored. Section 6 addresses the structure of QoS MIBs and Section 7 discusses how applications can access them. Section 8 addresses performance issues associated with QoS monitoring. Finally, Section 9 summarizes the main contributions of the architecture presented. A formal definition for the specification and measurement of QoS metrics is ommitted here due to space constraints and presented in reference [florissi95].

2 A Software Environment for Managing QoS Delivery

Figure 2.1 depicts the overall architecture of QoSME. The architecture proposed focus on QoS management as part of the runtime system, simplifying QoS handling for application developers. The horizontal lines divide the architecture layers depicted. The superimposed squares represent applications. The rectangles and the triangle represent functional modules. The tree shaped boxes represent a database. The straight arrows represent interactions between modules. The dashed arrows represent database access and updates by functional modules. At the application layer, the *Quality-of-service Assurance Language* (*QuAL*) provides abstractions for the negotiation of QoS constraints, specification of QoS violation handlers, and access to QoS MIBs. The QuAL compiler is responsible for translating these abstractions into calls to runtime components that allocate underlying system services and provide management of the QoS delivered. At the runtime layer, the *QoS for Sockets* (*QoSockets*) and the OS interface mitigate the interactions between QuAL applications and the underlying system components that deliver QoS demanding services. They provide a common OS and transport layer *Application Program Interface* (*API*), sheltering heterogeneity at the underlying system. For example, the same QoSockets interface is used for communication over TCP [comer91], ST-II [topolcic90], ATM [deprycker93], or any other protocol offered by the underlying environment. Similarly, the OS interface offers the same set of services independent of the underlying OS. QoSockets and the OS interface monitor interactions between applications and the underlying environment and update QoS MIBs with statistics on the QoS delivered to applications. A QoS SNMP agent embedded in QoSME runtime provides QoS MIB access to SNMP

[1] The reader is referred to [stallings93] and [rose93] for description of network management and the Simple Network Management Protocol (SNMP) standards.

managers, disclosing application QoS performance behavior to external management entities.

This paper focus on the main contributions inherent in the QoSME runtime components that assure QoS delivery on communications: QoSockets, QoS MIBs, and QoS SNMP agents. The design and novel features of QuAL and QoSME OS interface are discussed in [florissi94a, florissi94b] and omitted here due to space limitations. In a few words, QuAL consists of a language layer on top of the API offered by QoSockets and the OS interface. QoSME applications interact with QoSME runtime either through QuAL abstractions or through QoSockets and OS interface APIs directly. Even though the functionality offered by QoSME runtime is the same for both access methods, the QuAL compiler provides application development support that cannot be offered by QoSME runtime components. For example, the QuAL compiler can parse the specification of QoS constraints at compile time as opposed to runtime parsing performed by QoSockets functions.

QoSME OS interface bridges the gap between the services offered by the underlying OS and QoSME semantics for computing QoS. Consider, for example, QoSME running on top of Solaris. In QoSME, application activities must be scheduled based on their deadlines. Since Solaris does not provide this scheduling mechanism, QoSME OS interface mitigates interactions between applications and Solaris so that activities are scheduled on a earliest deadline basis.

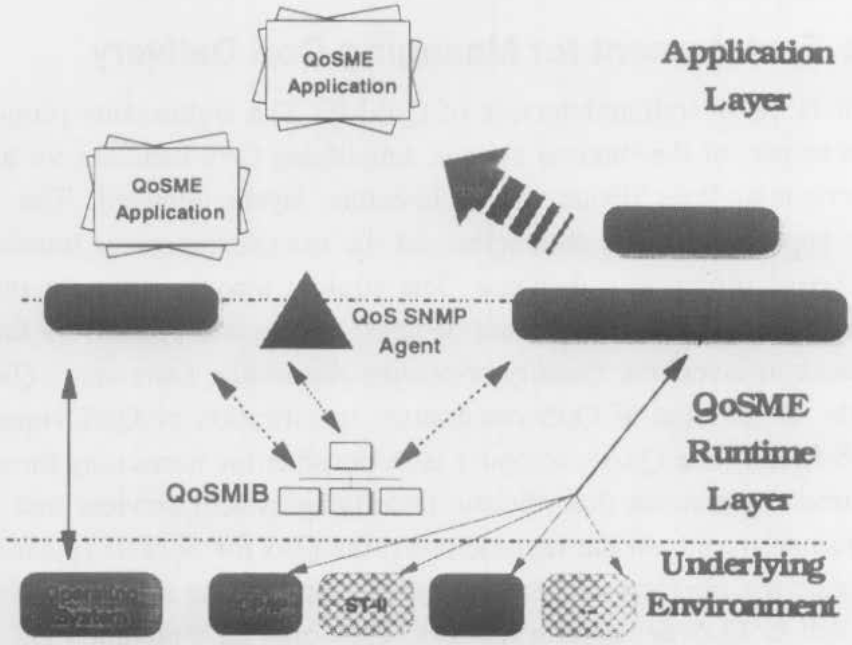

Figure 2.1: An Architecture to Manage Application Level QoS

QoSockets extends the sockets [stevens90] mechanism to include QoS negotiation, adaptation, and management by applications. Similarly to sockets, QoSockets abstracts a distributed system as a set of processes that exchange information through message passing. Messages are received through input ports (*inports* for short) and sent through output ports (*outports* for short). The term port is used when it is not necessary to distinguish an inport from an outport. QoSockets offers abstractions for attaching QoS constraints to ports and for negotiating QoS that simplify the development and maintenance of multimedia applications. QoSockets exposes

application developers to a single set of QoS abstractions for QoS negotiation between peer applications and between applications and the underlying environment. These abstractions are independent of underlying system configuration. QoSockets functions map abstract QoS specifications into runtime specific system calls.

QoSockets and the OS interface automate collection of application level QoS management information. These modules spy the interactions between applications and the underlying environment and reports to QoS MIBs statistics on the QoS delivered to applications. Examples of statistics collected are the number of messages delivered to a particular application connection and the average transmission delay of the messages delivered.

QoSockets applications adapt to QoS variations by assigning exception handlers to QoS violation events. QoSockets automatically analyzes QoS delivery and calls exception handlers when application defined violations are detected. A video application, for example, may use QoS constraints to specify the maximum acceptable jitter and an exception handler to adjust the playout time of frames when a violation occurs. Applications can further manage QoS by accessing QoS MIB values. QoSockets also offers a set of operators that provide real time local QoS MIB access to applications.

QoS MIBs and QoS SNMP agents provide the means to integrate application level QoS management into standard network management frameworks. Underlying system managers, for example, interact with QoS SNMP agents to monitor and analyze the delivery of QoS to applications, detect potential symptoms of QoS degradation, and control QoS violations.

3 How Should Applications Interact with QoS Delivery Mechanisms?

This section presents QoSockets, a software layer that mitigates interactions between applications and *QoS-supportive networks*. A QoS-supportive network [deprycker93] is a network that assures delivery of QoS metrics, in contrast to *non QoS-supportive networks* [comer91] that offer no QoS delivery guarantees. QoSockets extends sockets [stevens90] to support application interactions with QoS delivery mechanisms. Sockets creates a unified mechanism that permits applications and non QoS-supportive networks to vary greatly in design, but still be able to communicate through a single API. QoSockets aims at extending this concept to the domain of QoS supportive networks, by adding to sockets the ability to negotiate, adapt, and control QoS delivery.

There are several ways in which QoS-supportive networks may assure QoS delivery. Some approaches are discussed in what follows, while still others can emerge in the future.

Model 1. At one extreme, some networks [topolcic90, braden95] require that application end nodes explicitly instruct them about their QoS delivery needs and these networks configure end-end resources accordingly.

Model 2. At the other extreme, applications need not to negotiate QoS constraints. Networks [stevens90] monitor the execution of applications and adapt resource allocations dynamically, based on application behaviors. This approach is similar to OS memory management systems in which the OS monitors application memory accesses and adapts the allocation of physical memory pages accordingly. For example, TCP flow control mechanism [comer91] adapts buffer allocation for a communication according to application behavior.

Model 3. In an intermediate approach, networks [deprycker93] are configured to provide multiple virtual stacks and respective connectivities, each guaranteeing different QoS. End node management mechanisms select the appropriate stack over which a communication stream should be routed. This design can be seen as an extension of the ATM adaptation layer model [deprycker93] that offers four parameterized classes of services.

QoSockets aims at accommodating all of the QoS assurance models deployed by networks through a single mechanism for QoS assurance. The decision of which approach to use for an application or how to configure networks is beyond the scope of QoSockets and of this paper. QoSockets delegates to designers of transports and networks the choice of how QoS will be provided, and defers to application developers the selection of how QoS will be negotiated with the underlying environment. This section discusses how QoSockets supports all these approaches and bridge the gap between the QoS assurance model selected by applications and the model deployed by networks.

```
% Definition of a QoS Metric
typedef struct qos_metric {
        int value               /* Threshold for QoS metric */
        int window;             /* How often the QoS metric must be measured */
        int coercion;           /* If the threshold in value can be coerced at binding time */
} qos_met_ty;

% Definition of Universal QoS Metrics in QoSockets
typedef struct qos {
qos_met_ty loss;                /* Loss cannot be higher than 10-loss.value */
qos_met_ty permt;               /* A value higher than 0 in permt.value indicates that */
                                /* permutation is tolerated */
qos_met_ty average_delay;       /* End-end delay measured in ms must be */
                                /* lower than average_delay.value */
qos_met_ty jitter;              /* Jitter (inter-message arrival delay) measured in ms */
                                /* must be lower than jitter.value */
qos_met_ty min_rate;            /* Transmission rate measured in messages/s must be */
                                /* higher than min_rate.value */
qos_met_ty rate;                    /* Average transmission rate in messages/s must be */
                                /* lower than rate.value */
int size;                       /* Maximum message size */
int multiple;                   /* Port supports a maximum of multiple connections */
int combined;                   /* QoS metrics measure QoS on all multiple connections */
                                /* combined */
} qos_ty;
```

Definition 3.1: Specification of QoS Metrics in QuAL

In QoSockets, applications have the option to specify QoS metrics for a communication, as proposed in Model 1. Definition 4.1 specifies the *qos_ty* data type that enables the declaration of *universal* QoS metrics. Universal metrics are metrics that most applications need to negotiate and are used by the QoSockets runtime to allocate communicating and processing resources. For each metric, applications can specify a threshold value (field *value*), time intervals over which the metric should be measured (field *window*), and if the threshold can be *coerced* (field *coercion*) during binding time. QoSockets coercion mechanism will be discussed in Sec-

tion 4. QoSockets applications specify QoS constraints on a per port basis by associating a different *qos_ty* object with each port. For example, values 3 and 5 in the fields *value* and *window* of *average_delay* for a port *p* indicates that the *average_delay* QoS metric (formally defined in [florissi95]) cannot assume a value higher than 3 over intervals of 5 s on communications over *p*. QoSockets runtime uses *size*, when specified, to optimize resource allocation. Reference [florissi95] contains a detalied description of the functions supported by QoSockets to allocate ports, bind them, establish connections, and communicate data. They are ommited here due to space constraints.

QoSockets supports QoS negotiation on a multicast scenario. In QoSockets, ports support a maximum of *multiple* concurrent connections at a time. A positive value in *combined* indicates that the *min_rate* and *rate* QoS constraints refer to the rate of all the connections combined. When *combined* is not specified, each one of the *multiple* connections supports the *min_rate* and *rate* specified.

QoSockets applications can omit QoS specification for a communication, as proposed in Model 2. A *NULL* value for a *qos_ty* object field indicates that the application chooses not to specify that particular constraint and let up to the runtime to provide it. Thus, a *NULL* value for all *qos_ty* object fields indicate that an application chose QoS assurance Model 2.

QoSockets runtime bridges the gap between the QoS assurance model chosen by applications and the model deployed by a network. At one extreme, the network might require explicit specification of QoS constraints (Model 1) and applications might choose not to specify them (Model 2). In this case, QoSockets runtime firstly estimates an initial set of QoS constraints and request network services on behalf of applications. In addition, it automatically monitors the execution of applications and dynamically re-negotiates QoS with the network aiming at matching application behaviors with QoS delivery. Section 7 discusses how data collected during monitoring permits clever network management algorithms to adjust QoS delivery according to observed QoS metrics. At the other extreme, applications might choose Model 1 and networks might follow Model 2. In this case, the QoSockets runtime lets networks dynamically adapt to application behavior.

QoSockets shelters from application developers heterogeneity at the transport layer and below. QoSockets provides a single API for QoS specification that is independent of underlying transport mechanism specifics. QoSockets runtime translates abstract QoS specifications into transport specific service requests. Consider, for example, an application that chooses Model 1. If the underlying environment uses ST-II (Model 1), QoSockets runtime must map abstract QoS constraints in terms of rate and message size into specific ST-II buffer size parameters. If the underlying environment uses ATM adaptation layer (Model 3), QoSockets runtime maps application constraints into a service request for the class that best approximates application needs. Thus, it has significant advantages in relation to frameworks that are specialized for certain application domains [cohen81, cole81, keller93], and to approaches that expose programmers directly to transport layer and session layer service interfaces [topolcic90, anderson90]. Thus, QoSockets hides from applications the complexity of the underlying network services, including heterogeneity at the QoS assurance model.

4 How Should Peer Applications Negotiate QoS?

This section discusses QoSockets approach to support QoS negotiation between peer applications. Existing transport and application level protocols greatly differ on their mechanisms for this type of negotiation. In one paradigm [braden95, deprycker93, keller93], protocols make no provision for QoS negotiation between peer applications. One of the peer applications define QoS metrics for a communication and the other peer must comply with them. Applications must use out of band communications if they need to agree on the QoS of a connection prior to its establishment. Another paradigm [vogel94] suggests that applications receiving data publish a set of QoS classes that they can comply with and allow connecting applications to select one of the classes offered. Heterogeneity of QoS negotiation mechanisms may make code portability difficult because application code becomes tailored to meet the specifics of the protocols used.

The binding mechanism in QoSockets integrates QoS negotiation between peer applications into the sockets mechanism, while supporting existing negotiation paradigms. QoSockets abstracts QoS negotiation protocols as traditional language level type checking mechanisms. QoS constraints are part of the type of a port and QoS negotiation becomes part of assuring that the type of a port is not violated when ports are bound. Refecence [florissi95] presents QoSockets binding mechanism in greater detail.

```
1.  % Definition of Jitter Threshold. For a rate of 30 frames/s, jitter should not exceed 1/30s.
2.  define THOLD 1/30
3.  % Definition of a QoS Metric function
4.  double  video_jitter(qos_ppp *profile)
5.  {
6.      double j = 0;
7.      for (int i = 0; i < profile->size; ++i) {
8.          /* Check if jitter exceeded threshold. Assume there is no permutation or loss. */
9.          if(((profile->signatures[i+1].ta - profile->signatures[i].ta) * 1000 )> THOLD)
10.             ++j;
11.     }
12.     return j;
13. }
14. main()
15. {
16.     ...
17.     /* Trigger monitoring of video_jitter for inport rp */
18.     qos_monitor(video_jitter, 5, 1, rp);
19.     ...
20. }
```

Example 4.1: Monitoring Application Customized QoS Metrics in QoSockets

QoSockets supports several designs for QoS negotiation between peer applications. QoSockets binding mechanism guarantees that only ports with *compatible* QoS requirements are connected. Two ports have compatible QoS measures if QoSockets can *coarse* all the QoS requirements specified for the inport into the QoS requirements specified for the binding outport, or vice versa. For all constraints but *min_rate*, a coercion is possible when the QoSockets

can upgrade a less restrictive constraint until it matches a more restrictive one. For instance, QoSockets can upgrade an inport transmission delay of 4 ms into an outport transmission delay of 3 ms. *min_rate* cannot be coerced when an outport cannot deliver the minimum rate required by an inport. For example, an outport whose *min_rate* is 10 messages/s cannot be bound to an inport whose *min_rate* is 15 messages/s. Coercion of a QoS constraint is avoided when the field *coercion* assumes a negative value. In this case, the respective QoS constraint of an inport has to match exactly the constraint of a connecting outport, or they are not considered to be compatible. If only one application chooses Model 1, QoSockets runtime assumes the other side is able to comply with the constraints and allocates resources according to the QoS specified by one end. Ports that do not specify QoS constraints automatically have compatible QoS type.

QoSockets mechanism also hides from application developers heterogeneity on connection establishment protocols. Differences in QoS negotiation mechanisms employed by protocols lead to differences in connection establishment models. Some protocols [deprycker93] employ a *synchronous* model in which connection establishment requests block until the network and peer applications confirm that the QoS requested can be delivered. Other protocols [topolcic90] deploy an *asynchronous* model in which the underlying environment synchronously notify applications when negotiation terminates. Refecence [florissi95] discusses how QoSockets connection establishment protocol accommodates in a single protocol synchronous and asynchronous models.

```
% Definition of a Message Identifier
typedef struct mi {
        char *port_name;        /* Name of the port that originated the message */
        int index;              /* Index of the message in stream based on sending time */
} qos_mi;

% Definition of a Performance Signature
typedef struct pps {
        qos_mi  mi;             /* Message identifier */
        double[2] ts;           /* Message sending time measured in ms */
        double  ta;             /* Message arriving time measured in ms */
        double  ts;             /* Message processing time measured in ms */
        int     size;           /* Message size measured in number of bytes */
} qos_pps;

% Definition of a Performance Profile
typedef struct ppp {
        int size;               /* Number of performance signatures in profile */
        qos_pps* signatures[];  /* Array of performance signatures */
} qos_ppp;
```

Definition 4.1: Type Definition of Performance Profile in QoSockets

[2] In QoSockets, variables that indicate time are of type double because they store values of the sysUpTime objec [stallings93] maintained by the local management system. This object measures the number of milliseconds since the system was last initialized.

5 Automating Monitoring of Application Customized Metrics

QoSockets enables applications to program the QoS metrics that must be observed on their communications. Example 6.1 illustrates how an application receiving data can trigger monitoring of customized QoS metrics on the communication stream for its inport *rp*. Lines are numbered to ease referencing the code. It is assumed that *rp* is receiving video frames and that the application needs to know how many times the jitter was higher than a certain threshold. Lines 3 through 13 illustrate the definition of a *QoS metric function* in QoSockets. A QoS metric function is any function that takes a *performance profile* vector as input and returns a value of type float. The performance profile of a communication over a period of time *t* consists of the *performance signature* of all messages communicated during *t*. A performance signature of a message indicates its size and its sending, arriving, and processing times. Definition 6.1 shows the type definition of a performance profile vector. The application calls *qos_monitor* (line 18) to indicate that QoSockets must monitor the QoS metric *video_jitter* over time intervals of *5* s on the communication stream arriving on *rp*. The number *1* passed as third argument indicates that only one port is involved in the monitoring. QoSockets runtime automatically monitors the communication on *rp* and creates a performance profile for it. In addition, it calls the function *video_jitter* every time interval of 5s passing the performance profile for the last 5 s as argument to the function. The value returned by *video_jitter* is stored into QoS MIB entries. Section 6 discusses the architecture of QoS MIBs and shows where these values are stored. Section 7 illustrates how applications can retrieve QoS MIB values.

QoSockets enables the monitoring of QoS metrics that involve more than one communication stream. Consider, for example, a video conferencing application that must monitor how synchronized the video and audio streams are. This application passes 2 as the third argument to a *qos_monitor* call and the descriptors for the video and audio ports as forth and fifth arguments, respectively. QoSockets runtime will generate a performance profile that captures communication on both, the audio and the video ports. The application uses *port_name* in the *mi* field to distinguish signatures of the video communication from signatures of the audio communication, when necessary.

6 QoS MIB Architecture

A central challenge in the design of QoS MIBs is that they are programmable. QoSockets application developers can dynamically customize the QoS metrics that must be collected in QoS MIBs, as discussed in Section 5. This contrasts with current MIB designs where the instrumentation is rigidly determined at MIB design time and cannot be changed or adapted. Programmable MIBs are needed since QoS metrics vary according to application semantics.

Another challenge is that QoSockets applications and SNMP managers might need to access QoS MIB information simultaneously to manage QoS delivery. Thus, QoS MIB design must include a mechanism to coordinate management activities between them. At on end, applications manage QoS to adapt and recover gracefully from QoS degradations. At the other end, SNMP managers manage QoS to improve the performance of the underlying environment and to reduce QoS violation occurrences. Both activities can occur concurrently as long as they do not disturb eachother's effects. To address this issue, QoS MIBs contain *coordination* information, as discussed in the following subsections. When presenting a MIB object, the in-

formation stored in a group is classified in the following categories:

- *identification*: used to describe a particular instance of an object
- *configuration*: used to identify how resources were allocated for the service being monitored
- *operational behavior statistics*: used to analyze the actual performance delivered
- *coordination*: used to synchronize management actions between applications and SNMP managers

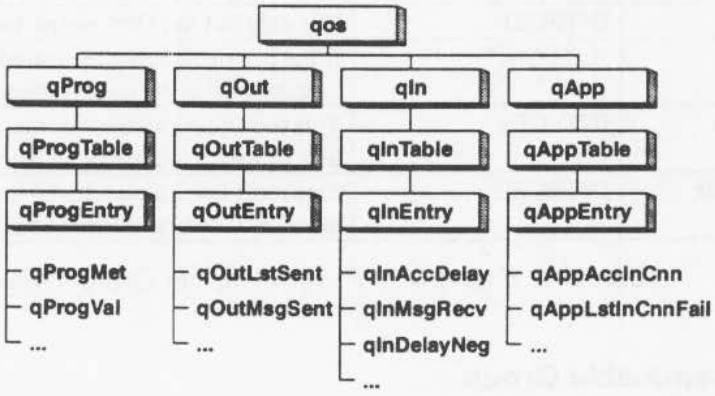

Figure 6.1: Overview of the Design of QoS MIBs

Figure 6.1 illustrates the overall QoS MIB structure that addresses the challenges above. QoS MIB data belongs to one of the following groups:

- *Programmable* (qProg[3], for short): consists of the table qProgTable which has one row of type qProgEntry for each application-programmed QoS metric being currently measured. Thus, entries are added to or removed from this group as applications trigger or cancel monitoring of new QoS metrics for a communication.
- *Outport (*qOut): consists of the table qOutTable which has one row of type qOutEntry for each QoSockets outport. Each entry indicates the value of the universal QoS metrics measured for an outport. Since, universal QoS metrics cannot be programmed, there is no need to create entries dynamically to store these metrics. Only one entry is created per outport, where each columnar object in the entry indicates the value of a metric measured on the outport side of a communication.
- *Inport* (qIn): is the equivalent of qOut for QoSockets inports. A qIn table extends a qOut entry to include the value of the umiversal QoS metrics that are measured on the inport side of a communication.
- *Application* (qApp for short): consists of the table qAppTable that contains one entry of type qAppEntry for each QoSockets application. Each entry indicates general information on the activities of an application. This group can be seen as an extension of the *Network Service Monitoring MIB* [freed93] (NSM MIB) to include information about application QoS.

[3] The name of QoS MIB objects starts with either qApp, qOut, qIn, or qProg depending on whether the object being named belongs to the application group, outport group, inport group, or programmable group, respectively. The prefix q indicates that they are related to QoS.

The following sections discuss each of these groups in greater detail and give examples of how their information can be used by applications and SNMP managers to manage QoS delivery.

#	Object	Syntax	Description
01	qProgMet	DisplayString	Name of the QoS metric programmed by an application.
02	qProgWindow	INTEGER	The size of the window over which the metric is measured
03	qProgLstTime	TimeStamp[4]	Last time when the metric was measured
04	qProgVal	INTEGER	The value of the QoS metric last time it was measured
05	qProgInOut	"in" | "out"	If the metric is being measured on the inport or in the outport side of the communication
06	qProgInAddr	INTEGER	Transport layer address of the inport of the communication in which the metric is being measured
07	qProgOutAddr	INTEGER	Transport layer addresses of the outport of the communication in which the metric is being measured

Table 6.1: Example of Programmable Group Objects

6.1 The Programmable Group

The goal of this group is to store QoS metrics programmed by applications. Table 7.1 illustrates some of the objects in this group. It is important to notice that based on the group objects 6 and 7, for example, QoS managers can trace the communication over which the QoS metric is being measured.

6.2 The Outport Group

Information in this group indicates the QoS requirements of outport connections, when specified by applications, and the values of QoS metrics measured during a communication. In addition, it includes information on connection problems and recovery performance. Identification objects store the local and remote IP addresses of the communicating machines, identification of the applications involved, and the transport layer port numbers of the connection. If a connection is currently presenting problems, managers use such objects to identify the applications involved and properly notify them. Similarly, if an application terminates abruptly, managers can look in the outport MIB for its connections and gracefully terminate them.

Table 7.2 shows some of the configuration objects present in the outport group. The qOutLoss, qOutPermut, qOutMinRate, qOutMaxRate, qOutPeak, qOutDelay, qOutJitter, qOutRecTime, and qOutMsgSize objects indicate the QoS metrics specified by applications, in case applications choose to specify them, or automatically assigned by QoSocekts runtime, otherwise. Managers use such objects to analyze the allocation of services to connections. Consider, for example, an application that receives radiology images and uses most of the communication resources on a machine. If other applications are unable to open connections, managers use qOutMaxRate, qOutPeak, and qOutMsgSize object instances to calculate how buffering resources are currently distributed. Managers will then realize that the amount of

[4] TimeStamp values store the value of the sysUpTime object maintained by the local management system at the time when the event being monitored last occurred. If the last occurrence of the event was prior to the last initialization of the local system, than the respective TimeStamp object contains a zero value.

bandwidth negotiated by the radiology application corresponds to a great percentage of the resources the machine has available. A manager might then force the radiology application to downgrade the QoS negotiated making possible for other applications to communicate concurrently.

#	Object	Syntax	Description
01	qOutProtocol	OBJECT IDENTIFIER	Identification of the protocol being used for this connection
02	qOutLoss	INTEGER	Probabilistic message loss rate ($10^{(-qOutLoss)}$)
03	qOutPermut	"yes" | "no"	Indication of tolerance to permutation
04	qOutMinRate	INTEGER	Minimum average number of messages per second
05	qOutMaxRate	INTEGER	Maximum average number of messages per second
06	qOutPeak	INTEGER	Maximum peak number of messages per second
07	qOutDelay	INTEGER	Maximum propagation delay
08	qOutJitter	INTEGER	Maximum inter message delay
09	qOutRecTime	INTEGER	Maximum time tolerated for recovery
11	qOutMsgSize	INTEGER	Maximum message size in number of bytes
12	qOutManager	OBJECT IDENTIFIER	Entity currently controlling communication QoS violations

Table 6.2: Example of Configuration Outport Group Objects

The qOutManager object indicates which entity is responsible for controlling communication QoS violations, enabling synchronization between application and SNMP management activities. Consider, for example, the case where an application sending video messages is experiencing a loss rate higher than expected. The video images being transmitted are of very high density and the intermediate nodes in the transmission path drop messages when there is not enough buffering space. In such case, the application may choose to reduce the loss rate by transmitting lower density images. qOutManager will indicate that the application is controlling the loss rate violation, inhibiting other managers from initiating any control action such as finding alternative paths for the communication.

#	Object	Syntax	Description
01	qOutCnnFail	Counter32	Total number of connection failures
02	qOutAccRecTime	INTEGER	Total amount of time spent in recovering
03	qOutActTime	TimeStamp	Time when the traffic became active
04	qOutMsgSent	Counter32	Total number of messages sent
05	qOutVolume	Counter32	Total volume of data sent in kilobytes

Table 6.3: Example of Operational Behavior Statistics Outport Group Objects

Table 7.3 illustrates operational behavior outport group objects. Objects such as qOutActTime, qOutMsgSent, and qOutVolume are used by domain managers to analyze how much of the resources allocated to an application are actually being used. A manager might force an application to reduce a communication allocation for a 30 frame/s video transmission to a 15 frame/s allocation, if the application has not sent more than 15 frames/s over a certain period of time. By detecting under-utilization, managers can re-distribute resources more efficiently.

6.3 The Inport Group

Table 7.4 illustrates some of for inport communications that enable the analysis of the transport level QoS actually delivered to an application. Thus, managers can directly compare the QoS being delivered with the QoS requested to service providers and trace the source of service degradations. For example, the display of video frames at a rate higher than human eyes can perceive (e.g., a display rate higher than 30 frames/s) may have been caused by violations on the maximum transmission and inter message delay. Such unexpected delays cause messages to arrive late and force the application processing them to consume messages in a rate higher than it should. The mean transmission delay of messages that arrive in sequence can be calculated by dividing qInAccDelay by qInMsgCounter. Similarly, the mean jitter can be calculated by dividing qInAccJitter by qInMsgCounter. At the same time, applications use such objects to adjust their performance according to the services provided, as discussed in Section 1. SNMP managers can also analyze the bandwidth distribution among connections by using inport group objects. The mean bandwidth usage for messages in sequence can be calculated by dividing qInMsgVolume by the difference between qInLstMsg and qInActTime. Domain managers can then control the distribution of resources based on such statistics.

6.4 The Application Group

The goal of this group is to store operational behavior statistics on the response of the protocol stack to QoS demanding connection establishment requests. Table 5.1 illustrates some of the objects in the application group.

Managers use application group objects to detect cases where application performance degradation is caused by poor transport layer resource allocation mechanisms. Consider, for example, an application that samples an audio device, detects silence periods, and transmits non silence samples. If such application is not scheduled during non silence periods, it will fail to capture pieces of the speech. Speech data will also be lost if the connection is lost and the application has subsequent connection establishments rejected. A manager uses qAppLstCnnFail to detect if connections were rejected recently. If that is not the case, the manager might decide to change the OS scheduling algorithm enabling the application to sample the device more often. If connections cannot be established, the manager might decide to re-distribute communication resources.

#	Object	Syntax	Description
01	qInActTime	TimeStamp	Time when the traffic became active, i.e., the first message was received
02	qInLstMsg	TimeStamp	Time when the last message was received
03	qInMsgCounter	Counter32	Total number of messages that arrived in sequence
04	qInMsgVolume	Counter32	Total volume of data received in kilobytes
05	qInAccDelay	Counter32	Total sum of the propagation delay of all messages that arrived in sequence
06	qInAccJitter	Counter32	Total sum of the inter message delay between any two consecutive messages that arrived in sequence

Table 6.4: Example of Operational Behavior Statistics Objects for Messages that Arrived in Sequence

#	Object	Syntax	Description
01	qAppLstCnnFail	TimeStamp	Time when the last connection to a QoS demanding inport was rejected
02	qAppManager	DisplayString	Entity currently managing QoS violations

Table 6.5: Example of Application Group Objects

The application group includes control configuration objects to coordinate management between applications and SNMP managers. For example, qAppManager indicates when a domain manager is changing the OS scheduling algorithm to allocate more processing resources to an application. An entity can only control a violation if another entity is not already doing so. Such constraint avoids chaotic situations where several entities are trying to solve the same problem without interacting.

```
1. main()
2. {
3. ...
4. /* Retrieve the number of messages received on rp */
5. msgRecv =
6. qos_snmp_get(qInAccMsgRecv.1234.128.59.25.32.4321.128.59.25.26);
7. ...
8. }
```

Example 6.1: Real Time QoS MIB Access Using SNMP Conventions

7 How Should Applications Access QoS MIB Values?

QoSockets design identifies three major challenges in supporting QoS MIB access by applications. The following sections discusses each one of the challenges and explain how QoSockets addresses them.

7.1 How Can Applications Access QoS MIB Data in Real Time?

Applications need to adapt to QoS delivery in real time. Thus, QoS MIB access to monitor and control QoS must be performed efficiently. In the SNMP framework, SNMP agents mitigate application accesses to MIB data. Once an application sends a request to an SNMP agent, there are no guarantees on when the request will be served. Thus, real time access is prohibitive in this scenario.

In order to provide real time QoS MIB access, QoSockets includes functions that enable applications to access the QoS MIB instrumentation directly, by passing SNMP agents. From the application point of view, these functions have the same semantics as SNMP get [rose93, stallings93] operations. However, these function offer real time response since they do not contend for a SNMP agent. Example 8.1 illustrates the use of the QoSockets function *qos_snmp_get* to retrieve the number of messages that arrived for inport *rp* from a given outport. qInAccMsgRecv is the name of the QoS MIB object that stores this type of information. According to the design of QoS MIB and SNMP standards, the instance of qInAccMsgRecv that stores information on *rp* is identified by the transport layer port address of *rp* (1234 in the

example), by the internet address of the machine where *rp*'s application is running (128.59.25.32), by the transport layer port address of the outport connected to *rp* (4321), and the internet address where the application communicating with *rp* is running (128.59.25.26). How applications find out the information needed to identify an object instance is outside the scope of this example. The example only illustrates that SNMP conventions can be used to access QoS MIB data in real time. The next section illustrates how QoSockets port descriptors, such as *rp* itself, can be used to identify QoS MIB instances. Thus, one possible way of finding out the transport layer address associated with a QoSockets port, for instance, would be to use the operators described below.

7.2 Which QoS MIB Objects Store Information on a QoSockets Port?

There is gap between QoSockets abstractions and the abstractions that must be used in SNMP to identify QoS MIB objects associated with QoSockets entities. In QoSockets, a port descriptor identifies a communication stream. In SNMP, however, lower level information, such as transport layer addresses, are used to identify QoS MIB objects associated with a QoSockets port. This gap makes it very difficult for applications to identify the objects that store information on a QoSockets port. For example, applications might need to find out the transport address of a port only to locate the QoS MIB objects associated with it.

```
1.  main()
2.  {
3.      ...
4.      /* Retrieve the sum in ms of transmission delays of all messages that arrived for rp */
5.      accDelay =
6.          qos_inport_get(rp, qInAccDelay);
7.      ...
8.  }
```

Example 7.2: Efficient QoS MIB Access Using QoSockets Abstractions

QoSockets offers a set of functions that automatically locate the QoS MIB entries associated with QoSockets ports. Example 8.2 illustrates the use of *qos_inport_get* to retrieve the sum in milliseconds of the transmission delay of all messages that arrived for *rp*. qInAccDelay identifies the object that stores this type of information. QoSockets runtime uses *rp* to automatically locate the instance of qInAccDelay associated with it and to retrieve its value. Similarly, QoSockets function *qos_outport_get* automatically locates QoS MIB objects associated with outports.

7.3 How Can Applications Be Notified of QoS Violations?

In order to detect QoS violations, applications monitor QoS delivery by constantly accessing QoS MIB data. This approach can highly disturb the execution flow of applications, specially if QoS violations are infrequent and traffic increases. Application code must be designed to periodically interrupt the flow of execution and jump into QoS monitoring mode. This context switching increases code complexity and might affect performance.

QoSockets supports automatic QoS violation monitoring. Applications inform QoSockets

runtime which conditions identify a QoS violation and the runtime performs the monitoring. Only when a violation is detected, the runtime notifies applications and the execution flow is interrupted to treat the abnormal event. QoSockets runtime notifies violations by sending notification messages to application defined ports. Example 8.3 illustrates the use of the function *qos_violation_monitor* to trigger handling violations of the QoS metric *video_jitter* defined in Example 6.1. As a result of the call, QoSockets runtime will automatically create a performance profile for inport *rp*, will call the QoS metric function *sync* every interval of *5s*, and send a notification message to inport *handler* every time *sync* returns a value lower than *2* or higher than *4*.

```
1.  main()
2.  {
3.      ...
4.      /* Trigger signaling of violations of the QoS metric sync measured for rp. */
5.      /* QoS metric sync must be measured every interval of 5s. */
6.      /* Notification messages must be sent to inport handler whenever */
7.      /* sync evaluates to a value lower than 2 or higher than 4 */
8.      qos_violation_monitor(rp, handler, sync, 5, 2, 4);
9.      ...
10. }
```

Example 7.3: QoS Violation Signaling in QoSockets

8 Challenges in Instrumenting QoS MIB

QoSockets architecture for monitoring and collecting information on QoS delivery aimed at fulfilling the following criterias:

- Monitoring and collection should incur minimal runtime overhead to preserve real time properties of QoS demanding activities. Data collection and access should not disturb the normal execution flow of applications. This aims at preserving the same behavior for an application, regardless of the underlying system architecture and of whether QoS is monitored or not.
- The information collected should be available concurrently to the application being monitored and to other applications involved in managing QoS delivery.

QoSockets adopted a *shared memory based design* that fulfills the criterias described above, as illustrated in Figure 8.1. The super imposed squares represent threads of execution of QoSockets applications. The cubes represent portions of shared memory, one for each QoS MIB group. The rectangular name tags indicate the QoS MIB group stored in a shared memory fragment, where the names qApp, qIn and qOut identify, respectively, the application, the inport, and the outport groups. The programmable group is omitted for simplicity, since data collection in this group is similar to the methods described for the inport and outport groups. Inside a thread of execution, the rounded rectangles represent *classes* of activities a thread can execute, such as *initialization* activities. The curved arrows indicate the execution flow of a thread shifting from one class of activities to another. The straight narrow arrows indicate the actions that a certain class of activities executes on the shared memory blocks.

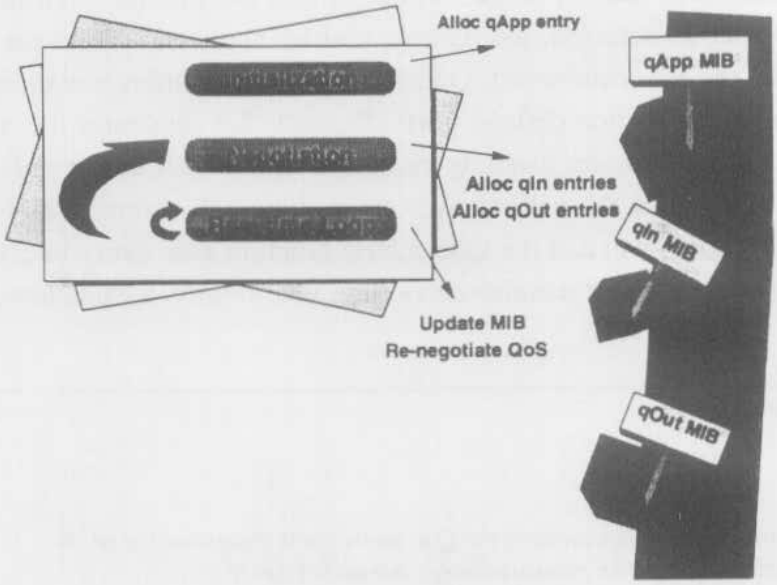

Figure 8.1: QoSockets Shared Memory Design for QoS MIB Data Collection

Memory allocation design in QoSockets is based on a general characterization of the execution flow of real time multimedia applications. The general characterization used classifies activities of such applications in three main groups:

- Initialization: includes start up activities, such as processing of initial arguments passed to applications.
- Negotiation: includes allocation of resources for engaging real time activities, such as establishment of QoS demanding connections and allocation of processing resources with the underlying *Operating System* (*OS*).
- Real time loop: involves the actual real time processing and transmission of data, such as sampling and sending of video frames.

Real time applications execute the real time loop until a task is completed or new QoS constraints are desired. In the later case, applications return to the negotiation phase, request new QoS, and engage the real time loop again.

To preserve real time properties of QoS activities, blocking operations are only performed when applications are out of the real time loop, either in the initialization or in the negotiation phase. The only blocking activity performed for QoS monitoring is the allocation of entries in the shared memory space that does not occur in the real time loop phase. Allocation of memory is blocking because an inter process synchronization mechanism must be used to coordinate entry allocation among QoSockets applications. Inter process synchronization can cause unpredictable delays and therefore would interfere in the execution flow of real time loops.

In QoSockets, QoS MIB updates do not affect the execution flow of real time activities. Shared memory updates do not require any synchronization among applications or between applications and QoS MIB SNMP agents. This is because QoS MIBs have no objects that can be written by more than one entity. Each application updates only specific fields of its own QoS MIB entries and these fields that can only be read by others. Similarly, QoS MIB SNMP agents have permission to write only in objects that cannot be written by applications. QoSockets allows, however, objects to be read or written concurrently. Thus, QoS MIB up-

dates in QoSockets have bounded computational cost, that is, the cost of a write operation in a shared memory position.

In QoSockets, QoS MIB data can be accessed concurrently by the QoSockets applications running on a system as well as by QoS SNMP agents that provide QoS MIB access to external managers. This is because QoS MIB data is stored in shared memory areas that are inherently accessible by all applications running on a system. However, shared memories are not robust to system failures. It is the responsibility of the QoS MIB SNMP agent to make backups of QoS MIB data and to garbage collect entries.

The QoSockets design for QoS MIB data collection is particularly suitable for multiprocessor architectures. In such architectures, shared memory blocks are also visible to the applications running in all processors. Because QoS MIB updates do not require synchronization mechanisms, applications can execute in real time simultaneously in distinct processors without interfering with each other for QoS monitoring. In addition, a processor can be dedicated only for the QoS MIB SNMP agent, in case the traffic of SNMP requests is very high.

9 Conclusions

This paper presents a software development environment for managing the *Quality of Service* (*QoS*) delivered to applications: *QoS Management Environment* (*QoSME*). In QoSME, applications interact with the underlying environment through *QoSockets*, an extension of the sockets mechanism that allows applications to negotiate, adapt, and control QoS delivery. QoSME automatically monitors QoS delivery and stores information collected during monitoring into *QoS Management Information Basis* (*QoS MIBs*). A *Simple Network Management Protocol* (*SNMP*) agent embedded in QoSME provides QoS MIB access to SNMP managers and applications. QoSME applications can also access QoS MIB through QoSockets functions.

QoSockets provides a uniform *Application Program Interface* (*API*) to support interactions between applications and the network QoS assurance mechanisms. QoSóckets supports several QoS negotiation models, ranging from explicit negotiation designs to fully automated ones. In the explicit negotiation model, applications inform the network about their QoS constraints. In the fully automated model, applications specify no QoS constraints while the network dynamically monitors application execution and adjust QoS delivery to match application behavior patterns.

QoS MIB instrumentation is completely automated. QoSockets runtime monitors QoS delivery and collects into QoS MIBs QoS metrics on communications. Applications can customize the QoS metrics measured dynamically. During execution, applications can define new metrics that must be monitored and instruct the runtime to monitor them. Applications can also define QoS violations and request QoSME to automatically signal when a violation occurs.

QoSME enables networks to monitor application needs and to incorporate this monitoring data into their resource management mechanism to support end-end QoS delivery. QoS MIB discloses to network resource managers information on the QoS needed by and delivered to applications. Thus, underlying system managers can adjust their resource allocation mechanism to best accommodate application needs.

A first prototype of the architecture proposed has been developed and released[5] by the DCC Lab. The first prototype runs on SunOS 4.3 and Solaris and supports communication on top of ATM, ST-II, TCP, UDP, and Unix local protocols.

References

[anderson90] Anderson, D. P., Tzou, S., Wahbe, R., Govindan, R., and Andrews, M., "Support for Continuous Media in the DASH System," in Tenth International Conference on Distributed Computing Systems, Paris, 1990.

[braden95] Braden, R., Zhang, L., Estrin, D., Herzog, S., and Jamin, S. "Resource ReServation Protocol (RSVP) -- Version 1 Functional Specification", Internet Draft draft-ietf-rsvp-spec-07, Integrated Services Working Group, July, 1990.

[cohen81] Cohen, D., "A Network Voice Protocol NVP-II," Tech. Rep., USC/Information Sciences Institute, April 1981.

[cole81] Cole, E., "PVP - A Packet Video Protocol," Tech. Rep., W-Note 28, USC/Information Sciences Institute, August 1981.

[comer91] Comer, D. E., Stevens, D. L., Internetworking with TCP/IP Volume 1. Prentice Hall, NJ, 1991.

[deprycker93] De Prycker, M., Asynchronous Transfer Mode: solution for Broadband ISDN, Second Edition. Ellis Horwood, 1993.

[florissi94a] Florissi, P. G. S., "QuAL: Quality Assurance Language (thesis proposal)", Tech. Rep. CUCS-007-94, Columbia University, March 1994.

[florissi94b] Florissi, P. G. S., and Yemini, Yechiam, "Management of Application Quality of Service", Fifth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Toulouse, France, October 1994.

[florissi95] Florissi, P. G. S., "QoSME: Quality of Service Management Environment (Ph.D. Thesis)", Tech. Rep. CUCS-036-95, Columbia University, December 1995.

[freed93] Freed, N., and Kille, S., "Network Service Monitoring Management Information Base", Internet Draft, November, 1993.

[halsall92] Halsall, F., Data Communications, Computer Networks and Open Systems. Addison Wesley, 1992.

[hyman93] Hyman, J., Lazar, A., Pacifici, G., "A Separation Principle Between Scheduling and Admission Control for Broadband Switching", IEEE Journal on Selected Areas in Communications, vol. 11, no. 4, 605 - 616, May 1993.

[keller93] Keller, R. and Effelsberg, W., "MCAM: An Application Layer Protocol for Movie Control, Access, and Management," in First ACM International Conference on Multimedia, Anaheim, 1993.

[lazar90] Lazar, A., Temple, A., and Gidron, R., "An Introduction for Integrated Networks that Guarantees Quality of Service", International Journal of Digital and Analog Communication Systems, vol 3, 229 - 238, April-June 1990.

[rose93] Rose, M.T., The Simple Book. Prentice Hall, 1993.

[stallings93] Stallings, W., SNMP, SNMPv2, and CMIP. Addison Wesley, 1993.

[stevens90] Stevens, W. R., UNIX Network Programming. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[topolcic90] Topolcic, C. "Internet Stream Protocol", Requests for Comments (RFC) 1190, October, 1990.

[vogel94] Vogel, A., Bochmann, G., and Gecsei, J., "Distributed Multimedia Applications and Quality of Service - A Survey," CASCON'94, Toronto, 1994.

[5] The URL address to QoSME release is http://www.cs.columbia.edu/dcc (go to projects and click on QoSockets). QoSME is also available by anonymous FTP to ftp.cs.columbia.edu. Login with the user name "anonymous" and a password of your e-mail address. After logging in, type "cd pub/qual", "binary", and then "get qual.tar.Z".