QoSME: QoS Management Environment

Patrícia Gomes Soares Florissi

Technical Report CUCS-036-95

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
1996

ABSTRACT

QoSME: QoS Management Environment

Patrícia Gomes Soares Florissi

Distributed multimedia applications are sensitive to the *Quality of Service* (*QoS*) delivered by underlying communication networks. For example, a video conference exchange can be very sensitive to the effective network throughput. Network jitter can greatly disrupt a speech stream. The main question this thesis addresses is how to adapt multimedia applications to the QoS delivered by the network and vice versa.

Such adaptation is especially important because current networks are unable to assure the QoS required by applications and the latter is usually unprepared for periods of QoS degradation. This work introduces the *QoS Management Environment* (*QoSME*) that provides mechanisms for such adaptation.

The main contributions of this thesis are:

- Language level abstractions for QoS management. The *Quality Assurance Language* (*QuAL*) in QoSME enables the specification of how to allocate, monitor, analyze, and adapt to delivered QoS. Applications can express in QuAL their QoS needs and how to handle potential violations.
- Automatic QoS monitoring. QoSME automatically generates the instrumentation to monitor QoS when applications use QuAL constructs. The QoSME runtime scrutinizes interactions among applications, transport protocols, and *Operating Systems* (*OS*) and collects in QoS *Management Information Bases* (*MIBs*) statistics on the QoS delivered.
- Integration of QoS and standard network management. A *Simple Network Management Protocol* (*SNMP*) agent embedded in QoSME provides QoS MIB access

to SNMP managers. The latter can use this feature to monitor end-to-end QoS delivery and adapt network resource allocation and operations accordingly.

A partial prototype of QoSME has been released for public access. It runs on SunOS 4.3 and Solaris 2.3 and supports communication on ATM adaptation layer, ST-II, UDP/IP, TCP/IP, and Unix internal protocols.

Table of Contents

List of Figures

List of Acronyms

AAL ..... ATM Adaptation Layer
API ..... Application Program Interface
ATM ..... Asynchronous Transfer Mode
CAT ..... Computer Assisted Test
ECS ..... Equipment Control System
EDF ..... Earliest Deadline First
FIFO ..... First In First Out
HWP ..... Heavy-Weigh Processes
IP ..... Internet Protocol
IPC ..... Inter-Process Communication
KLS ..... Kernel-Level Scheduler
LWP ..... Light-Weigh Processes
MCAM ..... Movie Control, Access, and Management
MIB ..... Management Information Bases
NSM MIB ..... Network Service Monitoring MIB
NV ..... Net Video
NVP ..... Network Voice Protocol
OS ..... Operating Systems
PEARL ..... Process and Experiment Automation Real-time Language
PVP ..... Packet Video Protocol
RTL/2 ..... Real Time Language/2
RPC ..... Remote Procedure Call
SAP ..... Service Access Points
SMI ..... Structure of Management Information
SNMP ..... Simple Network Management Protocol
SPS ..... Stream Provider System
SRP ..... Session Reservation Protocol
ST-II ..... Stream Transport Protocol Version II

TCP ..... Transmission Control Protocol
UDP..... User Datagram Protocol
ULS ..... User-Level Scheduler
QoS ..... Quality of Service
QoS-A ..... Quality of Service Architecture
QoSME..... QoS Management Environment
QoSockets ..... QoS in Sockets
QoSOS ..... QoS in OS
QuAL..... Quality-of-service Assurance Language
XDR ..... External Data Representation
XRM..... Extended Integrated Reference Model

Acknowledgments

I will always be in debt to the following:

- *God*, for His infinite love.
- *Prof. Yechiam Yemini*, my mentor, for making this thesis possible. His wisdom inspired me through all these years and his friendship made me survive the most difficult moments of my life. I can never thank him enough for giving me the privilege of being his advisee.
- *Danilo Florissi* (my best under graduate, masters, and PhD. colleague, my office mate, my best friend, and my husband), for inspiring and couching me in the last ten years, for always helping me, and for showing me what *love* really is.
- *My parents*, for teaching me to never give up, for always being there for me, and for their infinite love.
- *My family*, for their incredible support and love.
- *Susan Tritto*, for always being there to help and for her warm friendship.
- *German Goldszmidt*, for his friendship and for carefully reviewing this thesis.
- *My friends in the DCC lab*, for making me feel at home.
- *Mikhail Kishelev, Robert Shteynfeld, Margarita Safonova, Sanjay Jha, and all the students that worked in this research*, for their dedication and for showing me that together we can accomplish anything.

Chapter 1

Introduction

1.1 Background

Traditional network applications can operate under a broad range of network performance behaviors. They can tolerate very large end-to-end latency, accommodate greatly varying bandwidth, recover from loss, and endure dynamic fluctuations in latency and bandwidth. In contrast, distributed multimedia applications are very sensitive to the performance behavior of networks. A multimedia conference can be very sensitive to significant latency. Video streams require guaranteed bandwidth. Speech streams become incomprehensible under excessive jitter, i.e., dynamic fluctuations of latency.

It is therefore necessary to assure that the *Quality of Service* (*QoS*) performance[1] delivered by the network matches the one required by the applications and vice versa. The central question addressed by this thesis is how to develop mechanisms to accomplish this assurance.

Guaranteeing that the expected and delivered QoS match is not a simple problem. To start with, the relative sensitivity to QoS of multimedia applications often exceed by several orders of magnitude current network applications. For example, typical database or

[1] Appendix A formally defines QoS.

file server clients can tolerate jitter in packet arrivals ranging in many seconds. In contrast, a speech stream cannot tolerate jitter of several scores of milliseconds. This means that multimedia networks, in contrast with current networks, will need to assure delivery of the QoS required by applications. It also means that multimedia applications, in contrast with current network applications, will need to dynamically adapt to changes in the actual QoS delivered by the network. For example, an application may adapt to excessive jitter in a speech stream by increasing the play-out buffer size. An application may adapt to decreases in bandwidth available to a video stream by adjusting compression parameters.

Recent studies of QoS delivery have focused on the design of network mechanisms to assure QoS. These studies range from the design of *Asynchronous Transfer Mode* (*ATM*) [DePrycker 93] protocols and switching mechanisms to the design of multimedia transport protocols. Mostly, these mechanisms have focused on regulating competition for network resources among traffic sources. They involve resource allocation, flow, and admission control techniques used by packet/cell and transport layers.

This thesis uniquely focuses on the design of application-layer mechanisms to support effective adaptation to and control of QoS delivery. Specifically it addresses the following questions:

1. *How should applications adapt to the QoS delivered by the network?* Adaptation means first that applications can monitor the QoS delivery by the network transport. How can such monitoring be independent of the transport stack, to permit applications portability among different stacks? How can applications program the QoS metrics and events of interest to them? How can the instrumentation to

monitor QoS be generated without rendering applications design and implementation more complex? How can monitoring be accomplished without significant performance overheads?

Adaptation also means that applications must dynamically handle failures of the network to deliver the QoS that they require. How can applications incorporate constructs to handle QoS failures? How can this be accomplished without rendering applications design and code too complex.

2. *How can applications interact uniformly with a growing variety of network mechanisms to support QoS?* There is a growing variety of proposed techniques for network mechanisms to assure QoS delivery. One approach [Stevens 90] is to adapt the network delivery to specific applications needs. An application notifies the network of its QoS needs and the network configures its mechanisms to support appropriate delivery. Another approach [Topolcic 90, Braden et al. 95] permits applications to directly reserve network resources. Still another approach [DePrycker 93] is for the network to establish a small number of service classes and for the application to select among these the type of services that it wishes. Each of these models requires different interactions between applications and the network. In the absence of a uniform interaction model, applications will need to incorporate different code to interface with each of these mechanisms. This results in substantial complexity of applications design and limited portability across different networks. Furthermore, peer applications using different networks will have great difficulty coordinating their QoS control. A unified mechanism is

thus needed to negotiate and coordinate QoS between applications and the network.

3. *How can network management mechanisms monitor and control QoS delivery?* Emerging network architectures leave most error detection and correction functions to transport entities at end nodes. For example, loss or corruption of *Internet Protocol* (*IP*) [Comer and Stevens 91] packets will typically be detected by *Transmission Control Protocol* (*TCP*) [Comer and Stevens 91] entities at the end points and have serious impact on TCP performance. Similarly, loss of ATM cells will be primarily manifested in virtual circuit performance at the end points. Furthermore, protocol mechanisms to control performance at transport end points exhibit great sensitivity to such problems. For example, TCP dynamic window control mechanisms reduce window size dramatically in response to loss. This results in significant increase of latency and decrease in TCP throughput. Network management mechanisms must therefore monitor performance behavior of higher layers at end points to detect network problems. They must also be able to control the behavior of these end points. In contrast, current network management mechanisms have focused on monitoring and controlling lower layer protocol entities at intermediate nodes. They monitor and control the physical and routing/switching layers. There is thus a need to establish complementary mechanisms to monitor and control performance of transport delivery and applications at the end nodes.

This thesis introduces a new solution to these challenges: the *QoS Management Envi-*

ronment (*QoSME*). QoSME introduces application-layer technologies to address the questions above. These mechanisms permit applications to monitor and adapt effectively to QoS delivery by the network. They support a uniform model of interactions and control of QoS between peer applications and the broad variety of underlying network transport mechanisms. They facilitate monitoring and control of QoS delivery by network management systems. They hide the enormous complexity of QoS monitoring and control from the applications and their designers. They accomplish these with very minimal overheads.

This chapter is organized as follows. Section 1.2 outlines the core novel concepts introduced in this thesis. Section 1.3 overviews the QoSME architecture and places its components in the context of existing software environment architectures. Section 1.4 provides an overview of the reminder of the thesis.

1.2 Fundamental Principles

The goal of this section is to identify the novel concepts behind QoSME. They are:

1. *Language level abstractions for QoS management.* QoSME introduces novel programming language constructs to support applications monitoring, analysis and adaptation to QoS delivery. It also provides constructs that permit applications to convey their QoS needs to underlying computing and communication systems[2]. The main contributions of a language level approach to QoS management are:

   A. *Language level abstractions shelter applications and their designers from heterogeneity of the underlying systems.* Underlying networks and *Operating*

[2] From now on the expression *underlying system* will refer to the underlying network (with all the protocols up to and including the transport layer) and operating system.

Systems (*OSs*) vary greatly in the QoS control they offer. For example, the ATM *Adaptation Layer* (*AAL*) 3/4[3] allocates network resources to assure application specified throughput rates, while TCP makes no provision for such allocation. An application wishing to monitor and control its QoS will need to incorporate very different mechanisms to handle these transport environments. Application designers and their code will have to reflect the variety of possible underlying network mechanisms.

QoSME exposes application developers to a single set of abstractions for QoS management, independent of underlying system configuration details. The QoSME language compiler translates QoS specifications into specific service requests. This feature simplifies application development and maintenance, while increasing code portability and reusability.

B. *Applications can customize handling of QoS violations that affect them.* Applications differ on the type of QoS they are sensitive to. For example, an application receiving samples measured by a radar can be very sensitive to high transmission delays but insensitive to loss, since a radar typically sends several notifications of an event. On the other hand, distributed database applications may be very sensitive to loss but may tolerate high transmission delays. Adaptation to QoS delivery on a per application basis enables graceful, application customized recovery from QoS degradation.

C. *The language compiler checks many QoS specification inconsistencies.* For

[3] To simplify notation, AAL denotes AAL 3/4 in the reminder of this thesis.

example, the QoSME runtime reports on loss of video frames, as opposed to loss of ATM cells that contain only part of a video frame. In addition, it discards the remaining ATM cells of the damaged video frames. This feature simplifies QoS management by applications.

2. *Automatic QoS monitoring.* QoSME automatically generates instrumentation to monitor QoS. The runtime scrutinizes interactions among applications, communication protocol stacks, and OS, and collects into *QoS Management Information Bases* (*QoS MIBs*) [Rose 93, Stallings 93] statistics on the QoS delivered. It continuously checks the QoS constraints requested, detects QoS violations, and automatically invokes application exception handlers upon QoS degradations. This feature enables applications to dynamically adapt to the actual QoS delivered by the network. The main contributions of automated QoS monitoring are:

   A. *QoSME frees application developers from QoS monitoring instrumentation.* Application developers do not implement QoS MIB data collection and are thus sheltered from any complexity associated with this process. In addition, automation of QoS monitoring eliminates QoS MIB data inaccuracy introduced by programming errors in data collection procedures. This feature automates the implementation of performance statistics collection procedures.

   B. *QoS MIBs keep applications informed about QoS delivery performance.* Much like execution traces provided by debuggers, QoS MIBs disclose the performance of QoS demanding activities. Applications use QoS MIB data to dynamically adjust their execution according to the QoS being delivered, and to re-

quest additional QoS. For example, a video application can temporarily reduce the display rate to cope with network congestion. Simultaneously, it can open alternative connections that bypass congested routes. This feature simplifies the implementation of QoS adaptation procedures.

3. *Integration of QoS management within standard network management frameworks.* A QoS MIB *Simple Network Management Protocol* (*SNMP*) [Rose 93, Stallings 93] agent embedded in QoSME provides QoS MIB access to SNMP managers. The later can monitor end-to-end QoS delivery to applications and adapt network resource allocation and operations accordingly. This approach introduces the following contributions:

   A. *SNMP managers understand internal application QoS performance.* QoS MIB and SNMP agents inform SNMP managers about application QoS demands and the QoS effectively received. This information is more specific than generic performance statistics on network use. Resource managers use this data to improve the overall match between end-to-end QoS delivered by the network and each individual application needs. Also, they may identify problematic applications and terminate or isolate them. SNMP managers can thus get a more detailed picture on the QoS behavior of the network.

   B. *SNMP managers can share control of QoS performance with applications.* QoS MIBs provide the means to coordinate QoS management between applications and SNMP managers. This prevents chaos scenarios where QoS violation recovery actions by applications interfere with QoS management activities

from SNMP managers. This feature allows the partition of QoS management responsibilities between the network and the applications.

C. *QoS management can focus on application level properties.* QoS management may concentrate on application level abstractions (such as number of video frames delivered to an application) instead of system level objects (such as the number of bytes routed through a switch). This feature stimulates management of QoS independently of the underlying system operational details.

1.3 The QoS Management Environment (QoSME)

This section details the overall QoSME architecture depicted in Figure 1.1. The horizontal lines divide the architecture layers: the application layer, the QoSME runtime layer, and the underlying system layer. The superimposed squares represent applications and their threads. The rounded rectangles and the triangle represent functional modules. The tree shaped boxes represent QoS MIBs. The straight arrows represent interactions between modules. The dashed arrows represent QoS MIB accesses and updates by functional modules.

At the application layer, the *Quality-of-service Assurance Language* (*QuAL*) includes language level abstractions for QoS management (Item 1 in Section 1.2). Its constructs provide the means for the specification and negotiation of QoS constraints, specification of QoS violation handlers, customization of QoS monitoring, and access to QoS MIBs. The QuAL compiler translates these abstractions into underlying system services and provides management of the QoS delivered.

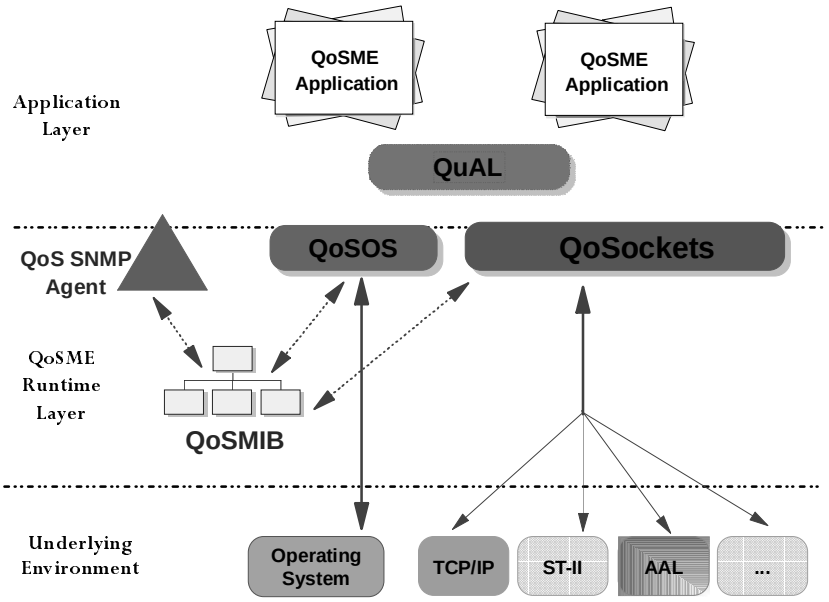

Figure 1.1: Architecture of QoSME

At the runtime layer, *QoS in OS* (*QoSOS*) and *QoS in Sockets* (*QoSockets*) mitigate the interactions between QuAL applications and the underlying system components that deliver QoS demanding services. They provide a unified OS and transport layer *Application Program Interfaces* (*APIs*), sheltering heterogeneity at the underlying system. The same QoSockets interface is used for communication over any transport protocol. Examples of such protocols are TCP, *Stream Transport Protocol Version II* (*ST-II*) [Topolcic 90], and AAL. Similarly, QoSOS offers the same set of services independent of the underlying OS.

The instrumentation to monitor QoS is generated automatically when applications that use QoSockets and QoSOS are compiled (Item 2 in Section 1.2). QoSockets and QoSOS monitor interactions between applications and the underlying system to update QoS MIBs with statistics on the QoS delivered to applications. They analyze QoS performance, de-

tecting QoS violations, and invoking application defined exception handlers when a violation occurs.

QoSME also integrates QoS management with standard network management frameworks (Item 3 in Section 1.2). The QoS SNMP agent embedded in the QoSME runtime provides QoS MIB access to SNMP managers.

In what follows, the major features provided by QoSME are identified in more detail. QuAL provides a small set of language extensions (to the Concert/C [Auerbach 92] process oriented [Hoare 78] language) that permits an application programmer to associate QoS measures and constraints with computations and communication streams. QuAL application developers are exposed to a single set of QoS abstractions that the compiler maps into runtime specific system calls.

The QuAL compiler generates code to monitor and analyze QoS delivery and to invoke exception handlers when QoS violations are detected. For example, a video application 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.

Upon QoS violation notification, applications may need to further investigate QoS performance to trace the cause of the event and control violations accordingly. QoS MIBs store the information needed to analyze such events. QuAL offers a set of operators that provide efficient local QoS MIB access by applications. For example, an application that is displaying video frames with a very poor performance usually checks QoS MIB data to analyze the cause of the service degradation.

QoSME abstracts in the QoSockets API the special transport services that support

communication QoS. QoSockets provide access to various transport protocols that support QoS, similar to the Berkeley socket [Stevens 90] mechanism. For example, a voice application may specify a low jitter 64 Kbit/s transmission using QoSockets. The later is translated in transport layer buffer and bandwidth allocation. QoSockets are a self contained module that can be used as a library on top of which new abstractions can be built.

QoSOS bridges the gap between the services offered by the underlying OS and the services required to implement QuAL abstractions for computing QoS. Consider, for example, the implementation of QuAL on top of Solaris [Sun Microsystems 94]. The semantics offered by QuAL abstractions is that processes are scheduled based on their deadlines. However, Solaris does not provide such scheduling mechanisms. In this case, the QuAL runtime OS interface is responsible for mitigating the interactions between QuAL applications and Solaris so that QuAL processes are properly scheduled.

QoSockets and QoSOS automate collection of application level QoS management information. These runtime components monitor the interactions between applications and the underlying system and store in QoS MIBs statistics on the QoS effectively delivered to applications. Examples of statistics collected are the number of messages delivered to a particular application level connection and the average transmission delay of the messages delivered.

Managers of underlying system resources 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.

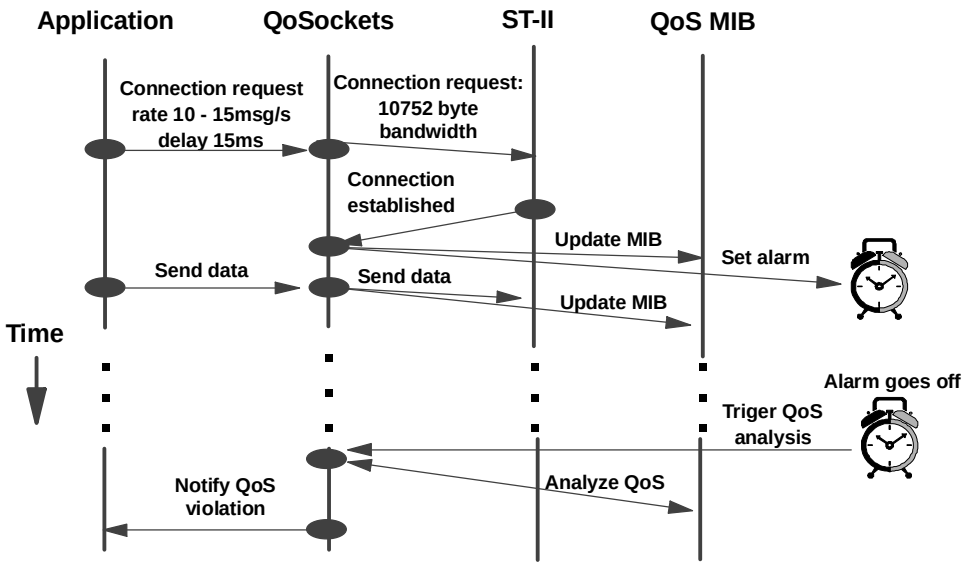

Figure 1.2: Timing of Events in a QuAL Connection

Consider, for example, a high resolution video transmission. Figure 1.2 shows the flow diagram of the interactions of QoSME components over time. An application first requests a connection and attaches to it QoS constraints. QoS constraints are expressed in QuAL by numerical intervals of tolerance. Example of such QoS constraints are a throughput of 10 to 15 messages/s and a 15 ms delay. The QuAL compiler translates this connection request into a call to the QoSockets services. QoSockets services are then responsible for choosing a transport protocol that can best deliver the QoS needed and for performing the QoS constraint mappings. In this example, QoSockets choose ST-II and translates throughput constraints expressed in messages per second into bandwidth requests expressed in bytes per second. Every time the application sends data through the connection, QoSockets deliver the data to the transport protocol and updates QoS MIB objects. The updates indicate, for instance, the mean number of messages sent per second. QoSockets also set alarms at application specified time intervals and blocks. Every time an alarm goes

off, QoSockets access the QoS MIB, analyzes the data retrieved, and sends a notification to the application if a violation is detected.

A first prototype of QoSME [Florissi 95] (QoSME 1.0) has been developed and released for public access. It includes the QoSockets module, the QoS MIBs, and the embedded SNMP agent. It runs on top of SunOS 4.1.3 [Sun Microsystems 92] and Solaris 2.4, and supports communication on top of AAL, ST-II, TCP, *User Datagram Protocol* (*UDP*) [Comer and Stevens 91], and Unix internal protocols [Comer and Stevens 91].

1.4 Thesis Overview

The focus of this thesis is on the design and use of QoSME. It is organized as illustrated in Figure 1.3, a derivative of Figure 1.1 where circles group QoSME components per thesis chapter.

Chapter 2 describes QuAL. It gives the informal semantics of the main QuAL constructs and shows examples on how they can be used.

Chapter 3 details QoSockets and QoSOS.

Chapter 4 describes the structure of QoS MIBs. It discusses in detail the type of information they store and gives examples on how QoSME applications and SNMP managers can use this data to manage QoS.

Chapter 5 describes applications developed using QoSME 1.0. It shows how the use of QoSME simplifies QoS handling in a few real applications and provides a preliminary study of the overhead and throughput of QoSockets.

Chapter 6 summarizes the main contributions of this work and suggests future re-

search topics not addressed in this thesis.

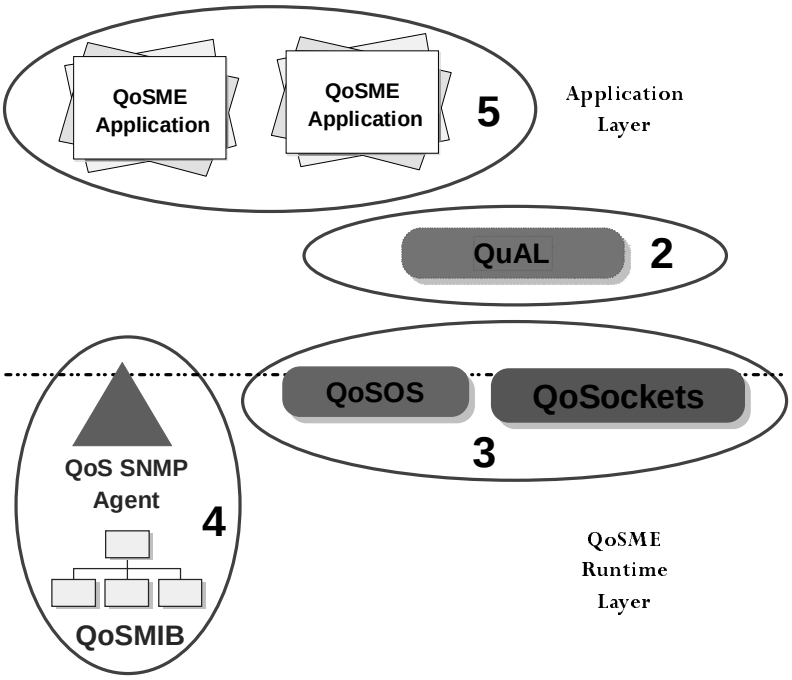

Figure 1.3: QoSME Components Grouped by Thesis Chapters

Appendixes A through *F* present a formal model for the definition of QoS metrics, overview Concert/C (a process oriented language that inspired QuAL abstractions), summarize the syntax and informal semantics of QuAL constructs, detail the specification of QoS MIBs, present related work on QoS management, and provide the Unix manual pages of QoSME 1.0.

Chapter 2

QuAL: Quality-of-service Assurance Language

2.1 Introduction

2.1.1 The Problem

The goal of the QuAL language is to enable application level management of QoS. It provides language level abstractions for application control of and adaptation to QoS.

QuAL constructs address the following challenges:

- *How to convey QoS requirements from applications to the underlying system?* Applications should be able to specify their QoS requirements in terms of abstractions that are independent of the underlying network and OS systems. For example, they should specify required inter video frame delays as opposed to inter ATM cell delays.
- *How to enable applications to adapt to QoS violations?* Applications need to know when the underlying system is not capable of providing the requested QoS in order to adapt their operations to the effective service received. They should specify the corrective measures that must take effect when QoS violations happen. For

example, if notified, applications may accept monophonic sound quality when the bandwidth is not wide enough to carry stereophonic sound. The corrective measure in this case specifies how to downgrade the audio quality from stereophonic to monophonic.

2.1.2 Main Results

QuAL views QoS as part of the data type associated with communication end points (or ports). Violations are analogous to invalid operation on these data types and recovery is similar to exception handling. QuAL provides constructs for the specification and management of QoS constraints. The main contributions of QuAL are:

- *Mechanism for QoS specification that shelters applications from the heterogeneity of the underlying network and OS systems.* QuAL provides a set of abstractions for QoS specification and negotiation that is independent of the underlying system details. The QuAL compiler and runtime system are responsible for mapping such abstractions into specific requests to the underlying system. The abstractions greatly simplify the development and maintenance of applications and promote code portability and reusability.
- *Mechanism to abstract QoS negotiation.* QuAL captures QoS constraints as part of the type of a port. The QuAL type checking performs QoS negotiations between peer applications by coercing port types to a common set of *compatible* QoS constraints before they are bound. QoS negotiation between applications and the underlying system happens when a connection is established. The QuAL type checking mechanism hides from application developers the complexity associated

with transport specific QoS negotiation protocols. It also integrates in a single mechanism QoS negotiation among peer applications, and between applications and the underlying system.

- *Support of application handling of QoS violations in real-time.* QuAL enables application developers to define QoS violation handlers. QuAL runtime automatically monitors QoS delivery and calls such handlers upon violations. This mechanism frees application developers from having to implement QoS monitoring while enabling application customized handling of violations.
- *Support of dynamic re-negotiation of QoS constraints.* QuAL provides a set of operators for QoS re-negotiation during run time. The QuAL runtime is responsible for handling re-negotiation details with the underlying system and for making the transition to the new constraints smooth. This mechanism enables applications to recover gracefully from QoS degradation, by gradually relaxing the negotiated QoS constraints.
- *Mechanism to integrate and coordinate QoS management between applications and SNMP managers.* QuAL applications are automatically instrumented at compile time to store the information collected during QoS monitoring into QoS MIBs. SNMP agents embedded in the QuAL runtime make QoS MIB data available to SNMP managers, disclosing application level QoS performance to the managers. This mechanism provides information that enables SNMP managers to adjust the allocation of system resources according to application needs.

2.1.3 Chapter Organization

The reminder of this chapter is organized as follows. Section 2.2 describes the main concepts behind process oriented languages and overviews the Concert/C language used in the design of QuAL. Section 2.3 and Section 2.4 discuss how QuAL supports the specification, negotiation, automatic monitoring, and violation detection of pre-defined and user-defined (or customized) QoS metrics. Section 2.5 explains how applications can control QoS performance. Section 2.6 presents the QuAL operators for access to communication temporal properties (e.g., sending and arriving time) of messages. Section 2.7 introduces abstractions for dynamic QoS re-negotiation. Section 2.8 overviews how QuAL integrates QoS management within the SNMP framework. Section 2.9 summarizes the main features of QuAL.

2.2 Process Oriented Languages

QuAL uses the *process model* [Hoare 78] to abstract the concept of distributed computations. *Processes* are units of execution that communicate and synchronize with one another through *message* passing. This section is an overview of the model. The reader may consult Appendix B for a more detailed presentation.

QuAL abstractions can be incorporated into any process oriented language. Concert/C was the choice for the first QuAL design. Concert/C extends the C language [Kernighan and Ritchie 88] to incorporate distributed computing abstractions. In Concert/C, a Unix thread [Sun Microsystems 94] in execution is a process and message exchange end-points are objects of type port. Messages are received through *input ports* (*inports* for short) and sent through *output ports* (*outports* for short). Outports are also known as *bindings*. An

inport is implemented as a queue of messages. A binding is implemented as a pointer to (the address of) an inport. The type of a port is determined by the type of messages it can exchange. Concert/C provides constructs for creating and terminating processes and for sending and retrieving messages through ports. The exchange of messages can be synchronous (implemented by *Remote Procedure Calls* (*RPC*) [Nelson 81, Soares 92]) or asynchronous (implemented by message passing). The design of Concert/C follows the Concert model [Yemini et al. 89, Auerbach et al. 91] for distributed computing.

The decision to prototype QuAL in Concert/C was motivated by its support for interprocess communication in C and by its runtime system design. Concert/C is completely compatible with ANSI C and adds to C a very concise set of data types and functions. Thus, for C programmers, the overhead of learning Concert/C is small. The implementation design of the Concert/C runtime is based on sheltering heterogeneity at the transport and OS layers, features shared by the QuAL runtime.

2.3 Handling Resource Level QoS Metrics

QuAL supports handling of two types of QoS metrics: *resource level QoS metrics* and *application specific QoS metrics*. Resource level QoS metrics provide performance measures of the underlying system in which an application operates. These include *universal communication QoS metrics* and QoS metrics related to computations. The universal QoS metrics are loss, permutation, rate, end-to-end delay, jitter, and connection recovery time (they are formally defined in Appendix A). QuAL runtime uses these metrics to allocate and manage necessary underlying system resources. Application specific QoS metrics indicate application dependent performance measures of communications. For example, a

video conference application may specify resource level QoS metrics such as *rate* and *delay* (as defined in Appendix A) to indicate how the runtime should allocate communication resources. In addition, it may define application specific metrics that indicate how synchronized its audio and video streams should be, that is, if each video and corresponding audio frames arrive at the same time.

Resource level and application specific QoS metrics differ also in purpose. Both specify how to perform QoS monitoring but only the first specifies allocation of system resources. For example, the *delay* metric specifies indirectly the strategy to allocate bandwidth and buffers while *rate of late messages* only specifies how to monitor the stream. It is left for future work mapping of application specific QoS monitoring into resource allocation strategies.

This section describes QuAL constructs for the specification, negotiation, monitoring, and violation detection of resource level QoS metrics. Section 2.4 describes QuAL constructs for handling application specific QoS metrics.

2.3.1 How Should Applications Interact with QoS Delivery Mechanisms?

There are several interaction models the underlying network system may use to assure resource level QoS delivery:

- *Explicit Model*. At one extreme, some networks [Topolcic 90, Braden et al. 95] require that applications explicitly specify their QoS needs in order to configure their resources. This is the approach used in ATM networks. For example, AAL requires explicit allocation of peak and mean rate tolerance per connection.
- *Implicit Model*. At the other extreme, applications need not allocate QoS con-

straints. Networks [Stevens 90] monitor the execution of applications and adapt resource allocations dynamically, based on application behaviors and resource availability. This is the approach used in the Internet. For example, TCP uses different mechanisms that automatically adapt to various traffic types, such as interactive traffic versus bulk transfers.

- *Intermediate Model*. In an intermediate approach, networks [DePrycker 93] are configured to provide multiple virtual protocol stacks and respective connectivity, each guaranteeing different QoS. The end node management mechanism selects the most suitable stack to route a communication stream. This design can be seen as an extension of AAL that offers four classes of services.

QuAL aims at accommodating these three models through a single mechanism for QoS specification and management. The decision of which approach a given application should adopt or how to configure networks is beyond the scope of QuAL and of this thesis. QuAL leaves to network protocols the choice of how QoS will be provided, and defers to application developers the selection of the model they want to employ. The remainder of this section discusses how QuAL bridges the gap between the model selected by applications and the one deployed by networks.

2.3.2 Specification of Resource Level QoS Metrics

QuAL provides a range of QoS attributes for the specification of network and OS resource level QoS metrics, following the Explicit Model. Consider, for instance, the scenario depicted in Figure 2.1 which involves a *Computer Assisted Test* (*CAT*)/scan at a remote hospital with real time analysis by physician A from a distant site. Application H at

the hospital samples images from the local CAT scanner and sends them to the display application PA at the physician's site.

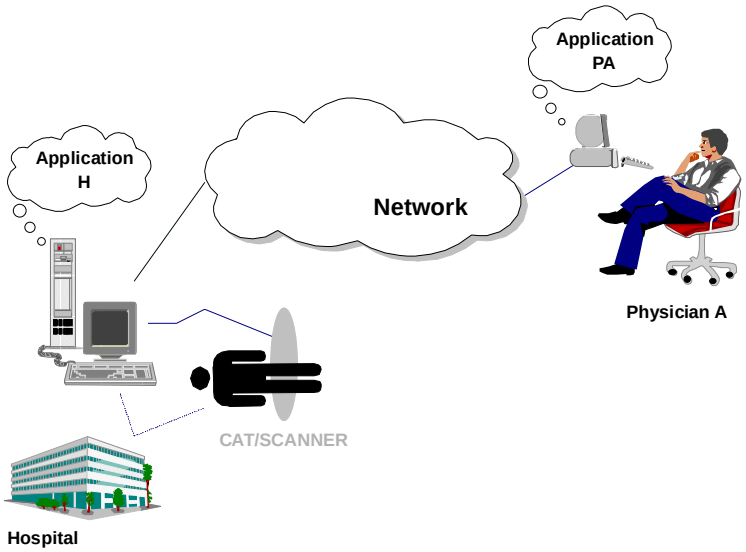

Figure 2.1: Remote Analysis of a CAT/scan

The following convention is used in the reminder of this chapter. Keywords and constructs in QuAL are written using ***bold*** face, in Concert/C are *underlined*, and in C are written using *plain text*.

Example 2.1 shows fragments of the QuAL code that implements application H in Figure 2.1. It defines the outport *image_output* (lines 3 through 14) that sends messages of type *image_t* (line 13). The main body (lines 18 through 32) consists mainly of a loop (lines 22 through 30) that periodically samples images from the scanner (line 27) and sends them through *image_output* (line 28).

QuAL extends the declaration of a port with the ***realtm*** (real time) clause (lines 3 through 11) that contains the specification of the universal QoS measures of interest for the port. For example, the delay QoS measure (line 9) indicates that the average delay of

messages sent through *image_output* must be lower than 40 ms. Using the model defined in Appendix A, a QoS violation occurs on *image_output* if $delay(\prod_{image\_output}) > 40$ ms. The QoS metric ***loss*** is the $loss_t$ defined in Appendix A, where *t* is the maximum delay tolerated on a communication. Permutation tolerance on *image_output* (***permt***) means that $permutation(\prod_{image\_output}) > 0$. QuAL runtime uses these metrics to allocate underlying system resources in order to prevent violations. The QuAL runtime only detects these violations at the end of the communication session. Section 2.3.4 discusses how applications can define time windows that force the QuAL runtime to monitor and detect violations over periods of time shorter than the duration of a session.

It is important to notice that the ***rate*** and ***peak*** metrics have the same semantics as far as monitoring is concerned. That is, according to the semantic model in Appendix A, both are evaluated over a window of time (in this case the duration of the connection) to check that the mean rate of messages in the communication is within the specified value. The difference in them is only apparent when allocating communications services. For example, the AAL resource allocating strategy may use a blending of both values to decide the buffer sizes to better accommodate eventual message bursts at the peak rate. Additionally, as explained later, an application developer may specify a smaller window to check peak rates and a larger one to check mean rates thus better approximating the correct semantics of these QoS metrics for monitoring purposes as well.

The QoS constraints specified in a ***realtm*** clause define the *QoS type* of the port. Because QoS is part of the type, QoS violations are considered invalid operations on the port. QoS violation handling will be further expanded in Section 2.3.4.

```
1 /* *** Application H *** */
2 /* Specification of QoS constraints on the transmission of CAT/scan images */
3 realtm {  loss 6;                  /* The percentage of messages lost during
4                                    transmission must not be higher than 10-6 */
5           permt;                   /* Permutation is allowed in the transmission, i.e.,
6                                    messages need not to be delivered in order */
7           rate sec 10 - sec 30;    /* Mean rate is 10 to 30 messages/s */
8           peak sec 35;             /* Peak rate is 35 messages/s */
9           delay ms 40;             /* Transmission delay must be less than 40 ms */
10          jitter ms 33;            /* Jitter must be less than 33 ms */
11          recovery sec 4;}         /* Any recovery must take less than 4 s */
12 receiveport                       /* Keyword that identifies a port declaration */
13   {image_t}                       /* Type of messages exchanged through the port */
14   *image_output;                  /* Port identifier */
15                                   /* "*" distinguishes an outport from an inport */
16 ...
17
18 main()
19 {...
20 /* Sampling and transmission of CAT/scan images */
21   /* Real-time loop */
22   within (                        /* Specification of timing constraints */
23             periodic; hard;       /* Mode of execution */
24             period sec 25;        /* Maximum number of activations */
25             after(machine_on));   /* Condition for first activation */
26   do {
27     read(scanner, &image);        /* Sample scanner */
28     send(image_output, image);    /* Transmit image */
29   }
30   until (machine_off);            /* Termination condition of real time loop */
31   ...
32 }
```

Example 2.1: Specification of Resource Level QoS Metrics

QuAL's real time language constructs enable expression of computing QoS constraints, such as deadlines to process messages. The ***within*** *real time loop* consists of a sequence of instructions (lines 26 through 29 in Example 2.1) and timing constraints associated with its execution (lines 23 through 25).

Execution timing constraints [Halang and Stoyenko 91] determine if an activity must be executed *periodically, sporadically,* or *aperiodically,* and can be either *hard* or *soft.* Periodic activities are those which have to be processed at regular intervals, and must be completed before the next one is due. Sporadic activities are asynchronous activities triggered by events. These events occur separated in time by a minimum guard period. That is, two event occurrence times are always separated by at least a guard interval. Aperiodic activities are asynchronous and have no minimum guard period between occurrences. Hard constraints must be met or else the application will deliver wrong results. Soft constraints indicate ideal targets which should be met when the system is not overloaded, but that can be missed occasionally. In Example 2.1, image sampling must execute in hard real time mode (line 23). The sampling must occur periodically (line 23) 25 times per second (line 24), producing 25 images/s.

The ***do*** block contains the task that must be performed in real time. It is terminated when the ***until*** condition is true. It executes for the first time only when the ***after*** expression is satisfied. In Example 2.1, sampling starts only after the patient is ready and the scanner is turned on (line 25), which is indicated by the event variable *machine_on*. Sampling continues until the scanner is turned off (line 30), which is indicated by the variable *machine_off.*

The specification of QoS constraints in QuAL is independent of the communication protocol, of the underlying OS, and of the nature of the data being transmitted (voice, audio, or data). This approach is general and broadens the domain of applications in which QuAL may be employed. Thus, it is more general than previous work that targeted spe-

cific application domains. For example, the *Packet Video Protocol* (*PVP*) [Cole 81] and the *Network Voice Protocol* (*NVP*) [Cohen 81] consist of transport layers customized for video and audio transmissions, respectively. PVP and NVP automatically negotiates with the underlying system the QoS needed for their video and audio transmissions. However, these protocols can only be used in the video and audio domains. Similarly, the *Movie Control, Access, and Management* (*MCAM*) [Keller and Effelsberg 93] is an application layer architecture customized for the handling of video streams in a distributed system. Appendix E contains a more detailed overview of related work in this area.

2.3.3 Negotiation of Resource Level QoS Constraints

This section discusses QuAL approach to support QoS negotiation between peer applications and between applications and the underlying system. Existing transport and application level protocols greatly differ on their mechanisms for negotiation. Some paradigms [Braden et al. 95, DePrycker 93, Keller and Effelsberg 93] make no provision for QoS negotiation between peer applications. One of the peer applications define desired QoS constraints and the other must comply with them. Applications must use out of band exchanges if they want to decide on a common QoS prior to connection establishment. Another paradigm [Vogel et al. 94] suggests that receivers publish a set of QoS classes that they can comply with and allow connecting applications to select one of the offered classes. Heterogeneity of QoS negotiation mechanisms may make code portability difficult because its logic becomes tailored to meet the particular mechanisms of the protocols used.

Instead, QuAL promotes symmetric negotiation by both sender and receiver which can

use QuAL constructs to agree on a common set of QoS. QuAL runtime uses such constraints to allocate communication resources.

QuAL abstracts QoS negotiation between peer applications by type checking connecting ports. The binding mechanism guarantees that only ports with *compatible* QoS attributes are connected. Two ports have compatible QoS measures if the compiler and runtime are able to *coarse* all the QoS requirements of the sender into the QoS requirements of the receiver, or vice versa.

Example 2.2 shows the fragment of application PA of Figure 2.1 that defines the QoS constraints associated with the reception of images from application H. It describes the specification of inport *image_input* (line 13) that has QoS constraints (lines 4 through 11) compatible with the constraints of outport *image_output* in Example 2.1.

For all constraints but ***rate*** (in special cases), a coercion is possible when the QuAL compiler (or runtime) can upgrade a less restrictive constraint until it matches a more restrictive one. For instance, the QuAL compiler can upgrade the recovery time of *image_output* (4 s) into the recovery time of the *image_input* (3 s). The compiler can also coarse the maximum delay tolerated by *image_input* (infinity, since it is not specified) into the maximum delay required by the *image_output* (40 ms).

The QoS attribute ***rate*** cannot be coerced when an outport cannot deliver the minimum rate required by an inport. For example, an outport with rate between 10 and 25 messages/s cannot be bound to an inport with rate between 15 and 30 messages/s because the outport cannot be forced to send more messages (15 per second) than its minimum (10 per second). When two ports have compatible ***rate*** constraints, the resulting ***rate***

interval for the communication is the intersection between the inport's and the outport's ***rate*** intervals. For example, the resulting ***rate*** interval for the communication between *image_output* and *image_input* is the interval between 10 and 25 messages/s.

```
1 /* *** Application PA *** */
2 /* Specification of QoS constraints on the reception of CAT/scan images */
3 /* Inport image_input will receive data sent by image_output in Example 2.1 */
4 realtm {  loss NULL;
5           permt NULL;          /* No constraints regarding loss or permutation */
6           rate sec 10 - sec 25;  /* Mean rate is 10 to 25 messages/s */
7           peak sec 30;         /* Peak rate is 30 messages/s */
8                                /* No constraints regarding delay */
9           jitter ms 33,        /* Jitter must be less than 33 ms */
10            nocoercion;        /* No coercion allowed in this QoS metric */
11          recovery sec 3;}     /* Any recovery must take less than 3s */
12  receiveport {image_t}
13    image_input;
```

Example 2.2: Negotiation of Resource Level QoS Metrics

Coercion of a QoS constraint is disabled by the keyword ***nocoercion*** in a QoS metric specification. In this case, the inport and outport QoS constraints are compatible only if they match exactly. Inport *image_input* does not allow coercion for jitter (line 10 in Example 2.2), but it can still be bound to *image_outport* because both port types match for this constraint.

Applications can comply to the Implicit Model by omitting QoS specifications. A ***NULL*** value for a QoS measure (lines 4 and 5 in Example 2.2) or the omission of a QoS measure all together (line 8) indicate that the application chooses not to specify that particular constraint. Chapter 3 explains how the QoSME runtime bridges the gap between

the model chosen by applications and the one used by the underlying network.

Inports may support multiple concurrent connections. Consider, for example, a geological seismic analysis application where a single inport needs to receive samples from several senders, as illustrated in Example 2.3. By default, a QuAL inport supports only a single connection. The keyword ***multiple*** (line 8) enables *samples_in* (line 13) to connect to a maximum of 10 ports concurrently. The keyword ***combined*** (line 9) indicates that the maximum ***rate*** (line 5) and ***peak*** (line 6) are measured over all connections combined. If the keyword ***combined*** were not specified, each connection would have to individually handle the specified QoS.

```
1 /* *** Geological Seismic Analysis Application *** */
2 /* Specification of inport that will receive data from several outports concurrently */
3 realtm {  loss 0;              /* No loss is tolerated */
4           permt ;              /* Permutation is allowed */
5           rate - sec 60;       /* Mean rate is at most 60 messages/s */
6           peak sec 80;         /* Peak rate is 80 messages/s */
7           delay ms 20;         /* Transmission delay must be less than 20 ms */
8           multiple 10,         /* Up to 10 outports can be connected to this inport */
9             combined;          /* concurrently and the combined mean and peak rate */
10                               /* of all the connections cannot exceed */
11                               /* 60 and 80 messages/s, respectively */
12 receiveport  {sample_t}
13   samples_in;
```

Example 2.3: Enabling Multiple Concurrent Connections to an Inport

QuAL's multiple port mechanism can be used to implement a multicast scenario, in which a single outport is connected to several inports. A message sent through an outport is received by all the inports connected to it. Multicast is enabled by the keyword ***multicast*** in an outport declaration.

QoS negotiation with the underlying network happens when a connection is established and it is based on the result of the coercion of the QoS demands of the communicating ports. For example, the runtime maps transmission rate constraints into protocol specific requests to allocate bandwidth and buffers.

The QuAL runtime automatically chooses the transport protocol that can best deliver the QoS needed. The choice of the communication protocol is delayed until run time, when two ports are actually bound. In this manner, the choice can be based not only on the static aspects of the communication (such as loss tolerance) but also on properties only known at run time. For example, properties such as what protocols are supported by the machines and network or whether communication will occur over a local or a wide area network can only be determined at run time.

QuAL runtime manages the OS resources available on a system and guarantees that applications engage real time execution only when their execution QoS demands can be satisfied. QuAL compiler estimates the *computational cost* of ***within*** loops and QuAL runtime performs a *schedulability analysis* of applications before they enter the real time mode of execution. The computational cost is the maximum amount of processor time required to execute sequentially on a dedicated uniprocessor the ***do*** block of a ***within*** loop. QuAL requires that the user provides a specification of a timeout period for every instruction or sequence of instructions whose computational cost cannot be estimated in advance, at compile time. Examples of this type of instructions are some loops, recursive function calls, and blocking instructions (such as reading and writing from a device). Timeout blocks will be further discussed in Section 2.3.4.

The QuAL compiler calculates computational cost in terms of the number and type of machine level instructions a QuAL block requires. The runtime then translates these costs in terms of execution time based on the specification of the architecture (which tells how long each machine level instruction takes to execute). The calculation of the computational costs provides an analysis of the worst case system load such that hard timing constraints are never missed.

The scheduleability analysis uses an heuristic procedure (discussed in Chapter 3) to verify if there is a sequential scheduling of QuAL application executions in real time mode such that they do not violate their timing worst case constraints given by the calculation of the computational costs.

The ***within*** loop initiates execution only if there are enough processing resources available to satisfy its constraints. In other words, only if it is scheduleable without disturbing the execution of other ***within*** loops. Otherwise, the runtime passes control to the statement following the ***until*** condition.

QuAL soft mode of execution permits to trade predictability for higher throughput. As opposed to what happens for hard real time blocks, soft real time blocks start execution even if it is known at compile time that timing constraints may not be met during overload periods.

In summary, the QuAL runtime maps QoS constraint specifications into OS system calls to allocate and manage the resources needed. For example, the runtime translates deadline constraints to execute a ***within*** loop into OS calls to reserve processing resources. The amount of resources requested and duration of the allocation are based on

the computational cost of the ***within*** loops and on its execution frequency.

2.3.4 Automatic Monitoring and Violation Detection of Resource Level QoS

QuAL runtime automatically monitors QoS delivery and invokes application customized exception handlers when QoS violations occur. It generates the *performance profile* of streams and initiates QoS performance control. The performance profile of a stream consists of the sending, arriving, and processing time of all messages in the stream (formally defined in Appendix A). QuAL runtime controls QoS performance as follows. First, the runtime uses performance profiles to calculate the value of communication QoS metrics during application defined time windows. Second, it checks whether or not a violation has occurred, that is, it checks if the measured metrics comply with the values specified for them. Finally, it calls the handlers when violations are detected. Similarly, QuAL runtime monitors application processing activities and initiates QoS control when application defined deadlines are missed.

Example 2.4 extends Example 2.2 to include specification of QoS management. It shows the updated declaration of *image_input* (lines 5 through 16) and fragments of the main body of application PA (lines 18 through 41). Application PA consists mainly of a real time loop (lines 22 through 39) that receives the images arriving on *image_input* (line 29) and displays them (line 30). The timing constraints (lines 23 through 26) specify a sporadic soft mode of execution (line 23), with a maximum of 25 activations per second (line 24). In this particular application, the designer decided that some frames may be displayed late and thus the soft execution mode suffices. Image displaying can be sporadically preempted by other QuAL applications that need to execute in hard real time mode. The exe-

cution must be sporadic the execution of the ***do*** block can only happen after an image arrives on *image_input* (lines 25 and 26). The ***after*** (line 25) expression causes the first execution of the ***do*** block to be suspended until the condition is true, that is, a message arrives on *image_input*. Message arrival on *image_input* is checked using the Concert/C function *select*. Similarly, subsequent executions of the ***do*** block are triggered by ***atEvent*** (line 26).

QoS performance on a communication is measured during application defined *time windows*. The declaration of the inport *image_input* is extended to include the specification of windows for the ***jitter*** and ***rate*** constraints (lines 5 and 7). The rate will be measured every 5 s and the jitter every 2 s. The default window size (used when it is not specified) is equal to the whole duration of the connection.

The QuAL runtime raises an exception every time a QoS violation is detected during time windows. It sends *exception messages* to application defined exception handler ports specified using the ***handlers*** clause (lines 10 through 15). An exception message indicates the time when the violation occurred and the type of violation (e.g., rate or jitter). For example, an exception message is sent to port *manage_conn* whenever the jitter is higher than 33 ms over a period of 2 s, or the rate is lower than 10 messages/s over a period of 5 s. Application PA is responsible for checking *manage_conn* and for adapting message display time upon violations.

```
1 /* *** Application PA (Extension of Example 2.2) *** */
2 /* Management of QoS on receiving CAT/scan images */
3 /* Inport image_input will receive data sent by image_output in Example 2.1 */
4
5 realtm {  rate sec 10 - sec 25, window sec 5; /* Mean rate must be measured every */
6                                               /* interval of 5 s */
7           jitter ms 33, window sec 2;         /* Jitter must be measured every */
8                                               /* interval of 2 s */
9 ...  }                        /* Specification of other QoS constraints */
10 handlers {                   /* Keyword that declares handler for */
11                              /* ports */
12   res_handler                /* Keyword that declares a handler port for */
13                              /* resource level QoS violations */
14        manage_conn;   }      /* Port that will receive notification of */
15                              /* resource level QoS violations */
16 receiveport {image_t} image_input;
17 ...
18 main()
19 { ...
20   /* Reception and display of CAT/scan images */
21   /* Real-time loop */
22   within (                                /* Timing constraints */
23        sporadic; soft;                    /* Mode of execution */
24        period sec 25;                     /* Maximum number of activations */
25        after(select(image_input));        /* Condition for first activation */
26        atEvent(select(image_input));)     /* Condition for subsequent activations */
27   do {
28        timeout(sec 1/40.0) {              /* Timeout block */
29          receive(image_input, &m);        /* Receive image */
30          ...                              /* Display image */
31        }                                  /* End of timeout block */
32        expired   { ...}                   /* Handle images that could not be displayed */
33   }                                       /* End of do block */
34   miss_deadline {
35     receive(image_input, &m);             /* Receive images that could not be displayed */
36                                           /* in time */
37     ...                                   /* Handle images that could not be displayed */
38   }
39   until (false);                          /* Termination condition */
40 ...
41 }
```

Example 2.4: Detecting Resource Level QoS Violations

The QuAL runtime also raises exceptions when processing resources cannot deliver the QoS requested to execute real time blocks. QuAL supports the notion of a *frame* associated with a ***within*** loop. A frame is the maximum duration allowed for a loop activation. For example, if the maximum number of activations of a ***within*** loop is 25 times/s, as in Example 2.4 (line 24), the frame associated with this loop is 1/25 s. Assuming a worst case scenario in which all activations do occur, QuAL assumes that each loop execution of a ***within*** statement cannot take longer than the duration of a frame, otherwise an exception handler must be invoked. In Example 2.4, the execution of the ***do*** block (lines 27 through 33) is interrupted and control passed to the ***miss_deadline*** exception handler (lines 34 through 38) whenever its execution does not reach completion within a frame duration.

It is important to notice that the QuAL runtime starts counting the execution time of a ***do*** block from the moment the block is eligible to start (e.g., upon message arrival on *image_input*) and not from the moment the block actually starts executing. Thus, the ***miss_deadline*** handler is invoked whenever the ***do*** block does not complete execution within 1/25 s of a message arrival in *image_input*.

The ***timeout*** block can be used to specify explicit execution deadlines for instruction blocks with computational cost that cannot be calculated at compile time. The main goal is to assign a maximum execution time for the instruction block and to interrupt it if its execution takes longer than predicted. Assume that in this particular example, it cannot be known at compile time how long the display operation will last. A ***timeout*** *block* (lines 28 through 31) is used to specify the maximum image processing duration. That is, either the

display is executed in 1/40 s or an exception is raised, interrupting the execution and passing control to the ***expired*** block (line 32). The code in the ***expired*** block may decide to handle the violation by discarding the offending frame.

QuAL contains a general purpose high level real time language suitable for hard and soft real time programming with predictable behavior. QuAL schedulability analysis and exception handling mechanisms are applicable for hard as well as soft activities. Most soft real time languages do not support hard real time constraints and implement a more restrictive exception handling mechanism. Examples of this type of languages are *Real Time Language/2* (*RTL/2*) [Barnes 76], *Process and Experiment Automation Real-time Language* (*PEARL*) [Kappatsch 77], ILIAD [Pickett 79], PORTAL [Nageli and Gorrengourt 79], and Ada [Ada 83]. Furthermore, with the introduction of the ***timeout*** block, QuAL permits the use of traditional C constructs without sacrificing behavior predictability.

Chapter 3 discusses the implementation of QuAL constructs and Appendix E includes a more detailed account of related work in real time programming languages.

2.4 Monitoring Application Specific QoS Metrics

The QuAL runtime also automates monitoring of application specific QoS metrics. An example of such QoS metric is the rate in which an application is sending messages through an inport (which may differ from the ***rate*** metric defined in Section 2.3). This section first discusses how applications can specify these QoS metrics. Then it shows the mechanism in QuAL runtime to signal applications if they are violated.

```
1 /* Definition of a threshold for the difference between the arriving time of
2    consecutive messages. At 30 frames/s, the difference must not exceed 1/30 s. */
3 #define THOLD 1/30
4 /* Definition of a QoS Metric function */
5 double inter_arrival_delay(double[1] start, double end, qos_ppp *profile)
6 {
7    double j = 0;
8    for(int i = 0; i < profile->size; ++i) {
9       /* Check if difference between arriving times exceeded threshold. */
10      /* Assume there is no permutation or loss. */
11         if(((profile->signatures[i+1].ta - profile->signatures[i].ta) * 1000 )> THOLD)
12            ++j;
13      }
14      return j;
15   }
16   main()
17   {
18      ...
19      /* Trigger monitoring of inter_arrival_delay for inport video */
20      qual_monitor(inter_arrival_delay, 5, 1, video);
21      ...
22   }
```

Example 2.5: Monitoring Application Specific QoS Metrics

2.4.1 Automating Monitoring of Application Specific QoS Metrics

Example 2.5 illustrates how a video conference application can trigger monitoring of application specific QoS metrics on its inport *video*. The inport *video* is receiving video frames and the application needs to know how many times the difference between the arriving time of consecutive messages was higher than a certain threshold. This metric will be called *inter_arrival_delay*. Lines 4 through 15 illustrate the definition of a *QoS metric function* in QuAL. A QoS metric function is any function that returns a value of type

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

double and takes as input the start and end times of a window and *performance profile vectors*. Definition 2.1 shows the type definition of a performance profile vector of a stream. It consists of a collection of the *performance signature* of the messages in a stream. The performance signature of a message consists of the sending, arriving, and processing time of the message (formally defined in Appendix A). Performance signatures in a vector are ordered either by sending time or by arriving time, depending on whether they are generated on the outport side or on the inport side of a communication. In the example, the profile passed to *inter_arrival_delay* will be ordered by arriving time because *video* is an inport.

```
/* Definition of a Performance Signature */
typedef struct pps {
  double ts;                  /* Message sending time measured in ms */
  double ta;                  /* Message arriving time measured in ms */
  double tp;                  /* Message processing time measured in ms */
} qos_pps;

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

Definition 2.1: Type Definition of a Performance Profile Vector

Applications call the function ***qual_monitor*** to initiate QoS monitoring. In the example, the call to ***qual_monitor*** (line 20) indicates that the runtime must monitor the QoS metric *inter_arrival_delay* arriving on *video* over windows of 5 s. The number 1 passed as third argument indicates that only one port is involved in the monitoring. The runtime

automatically monitors the communication on *video* and generates a performance profile for it. In addition, it calls the function *inter_arrival_delay* every 5 s passing the start and end times of the last window and the performance profile collected during it as argument to the function. The value returned by *inter_arrival_delay* is stored into QoS MIB entries. Section 2.8 discusses the architecture of QoS MIBs, where these values are stored, and how applications can retrieve them.

```
1 /* Definition of QoS metric to measure audio and video stream synchronization */
2 double synchronization(double start, double end,
3                         qos_ppp *video_profile, qos_ppp *audio_profile)
4 {
5   ...
6 }
7 main()
8 {
9   ...
10      /* Trigger monitoring of synchronization for inports video and audio */
11    qual_monitor(synchronization, 5, 2, video, audio);
12    ...
13  }
```

Example 2.6: Monitoring of Group Application Specific QoS Metrics

QuAL allows monitoring of group application QoS metrics. QoS metric functions can take several profile vectors, one for each communication stream monitored. Consider, for example, the case when the application in Example 2.5 needs to measure how synchronized the streams arriving on *video* and *audio* are, as illustrated in Example 2.6. The call to ***qual_monitor*** (line 11) now passes the number 2 as the third argument, indicating that two streams need to be monitored. In addition, a fifth argument is added to the call indicating that *audio* is the second port to be monitored. The QuAL runtime monitors both

streams and generates a performance profile for each one. After each window, the runtime calls the function *synchronization* and passes the start and end times of the window and the performance profiles of the inports *video* and *audio* as the third and fourth arguments, respectively. There are no design imposed limits to the number of ports that can be monitored for a particular QoS metric.

2.4.2 Automatic Notification of Application Specific QoS Violations

QuAL supports automatic QoS violation monitoring. Applications inform the QuAL runtime which conditions identify a QoS violation and, when a violation is detected, the runtime notifies the applications. QuAL runtime sends notification messages to application defined ports, similar to what happens with resource level QoS violation detection.

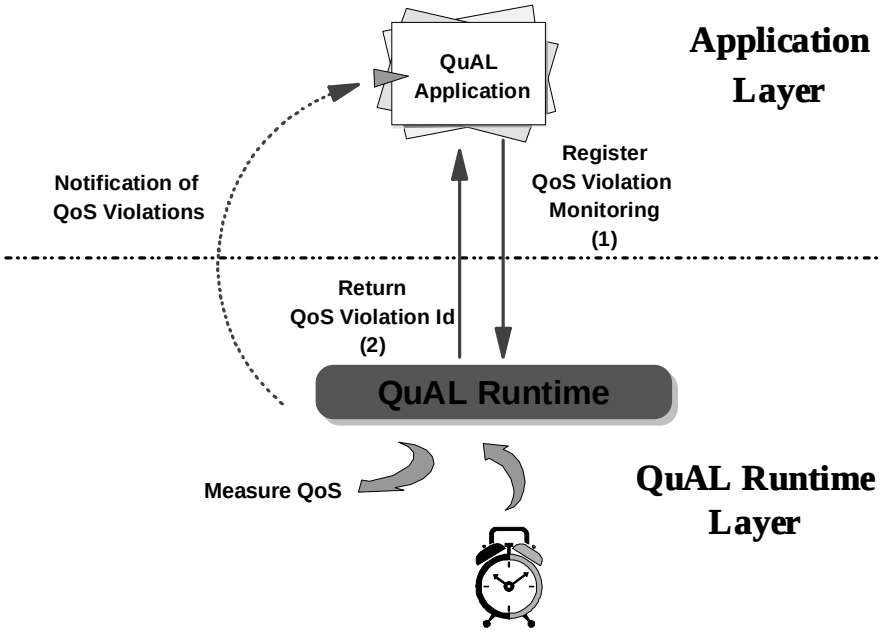

Figure 2.2: Detecting QoS Violations

Figure 2.2 depicts the architecture of the mechanism used by the QuAL runtime to

provide QoS violation signaling. The narrow straight arrows indicate information flow between applications and the runtime through system calls. The dashed arrow represents the flow of messages from the runtime to application inports, notifying QoS violations. The small full triangle represents an application inport. Applications register with the run-time QoS violations that must be monitored. As a result of the call, the runtime returns a QoS violation identifier. Upon registration, QuAL runtime sets clock alarms according to the windows specified by the application and analyzes QoS performance every time an alarm goes off (curved wide arrows).

```
1 /* Requesting notification of application specific QoS violations */
2 /* QoS metric inter_arrival_delay was defined in Example 2.5 */
3 main()
4 {
5   /* Trigger signaling of violations of inter_arrival_delay measured on inport video. */
6   /* Notification messages must be sent to inport handler whenever inter_arrival_delay
*/
7   /* is lower than 2 or higher than 4 */
8   id = qual_violation_signalling(inter_arrival_delay, 5, 2, 4, handler, 1, video);
9   ...
10  }
```

Example 2.7: Signaling Application Specific QoS Violations

For example, QoS violation signaling is particularly suitable for play-out time control of video images. An application displaying video at 30 frame/s may need to detect when the video transmission jitter (as defined in Appendix A) is higher than 1/30.0 s over a given window (e.g., half a minute). If the mean jitter is higher than 1/30.0 s, then not enough frames will be available for a 30 frame/s display during the next window. The application must then gracefully reduce the display rate by gradually increasing the inter mes-

sage display interval and discarding late messages. The application registers the violation event with the runtime. Execution is only interrupted when the runtime notifies the application that the event has indeed occurred and not after every window.

Example 2.7 illustrates the use of the function ***qual_violation_signalling*** to trigger the monitoring of violations to the QoS metric *inter_arrival_delay* defined in Example 2.5. As a result of the call (line 8), QuAL runtime automatically creates a performance profile for inport *video*, calls the QoS metric function *inter_arrival_delay* every interval of 5 s, and sends a notification message to inport *handler* every time *inter_arrival_delay* returns a value lower than 2 or higher than 4. The function call returns an identifier for the violation registered.

```
/* Definition of violation types */
enum {min, max} qual_vltn_ty;   /* Violations either on min or max thresholds */

/* Definition of the type of a notification message */
typedef struct {
  int id;                       /* Identifier of the QoS metric that was violated */
  qual_vltn_ty vltn_ty;         /* Type of the violation */
  double sample;                /* QoS metric value that triggered violation */
  double time;                  /* Time when the violation was detected */
  } qual_violation_ty;
```

Definition 2.2: Type of a Message Notifying a QoS Violation

Definition 2.2 shows the specification of a notification message which indicates which QoS violation was detected, the type of the violation (i.e., minimum or maximum threshold violation), the value of the QoS metric when the violation was detected, and the time of the occurrence.

2.5 Specifying Filters

This section describes QuAL constructs for the specification and negotiation of *filters*. A filter is a function that modifies the performance profile of a stream (as defined in Appendix A). Filters allow applications to control QoS performance. For example, consider a sender that is generating more messages than its receiver is capable of processing. These applications can use filters to agree under which circumstances (e.g., sending time or current system load) a message generated should be inserted in or discarded from the stream. Filters provide protection similar to the one provided by a fuse when installed in an electrical circuit. That is, it preserves applications from damaging whole systems when logical glitches happen. In the example mentioned, control of the sender rate may prevent receiver buffer overflow with consequent connection or application failures.

2.5.1 Specification of Filters

Filters are inspectors placed in ports to check the data flow, that is, to guarantee that only complying messages are injected in the communication stream. They can analyze flows generated for an outport or arriving on an inport, as illustrated in Figure 2.3. The dashed horizontal lines divide the three layers depicted: the application layer, QuAL runtime layer, and the underlying system. The arrows show the data communication path between two remote applications, A and B. The two inspectors check each message as they are about to leave or enter the application layer.

In the QuAL model of communication, messages that do not comply with the metrics enforced by the filters (the dashed lines in Figure 2.3) are either discarded or forwarded to exception handler ports, depending on how ports are specified. Alternatively, filtering can

simply trigger exception messages, without actually removing messages from the stream. This feature will be further explored in Section 2.5.3.

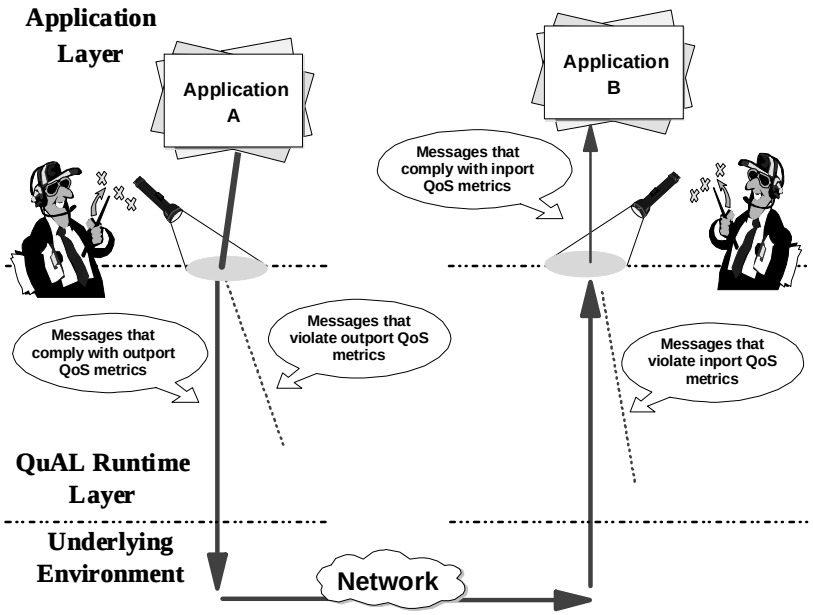

Figure 2.3: Adding Filters to Manage QoS Performance

Figure 2.4 illustrates an extension of the CAT/scan application in Figure 2.1 for broadcast of images on an heterogeneous system. A technician joins the exam team to help to operate the scanner. The technician verifies that the patient and the scanner are correctly positioned and that the images are clear. Application T running on the technician's computer is responsible for processing the messages sent by application H. The technician is using a mobile computer and has less computing and communication resources than Physician A. Therefore, she is not able to receive images at the same rate as the physician is. In reality, the technician does not need to receive every single image in order to calibrate the scanner. It is enough for the technician to only look at half of the images generated, since patient positioning and image clarity are unlikely to change on a per image basis. In

this scenario, filters are used to guarantee that only half of the messages generated by Application H are actually sent to Application T, tuning the interaction between applications H and T according to the resources available to them.

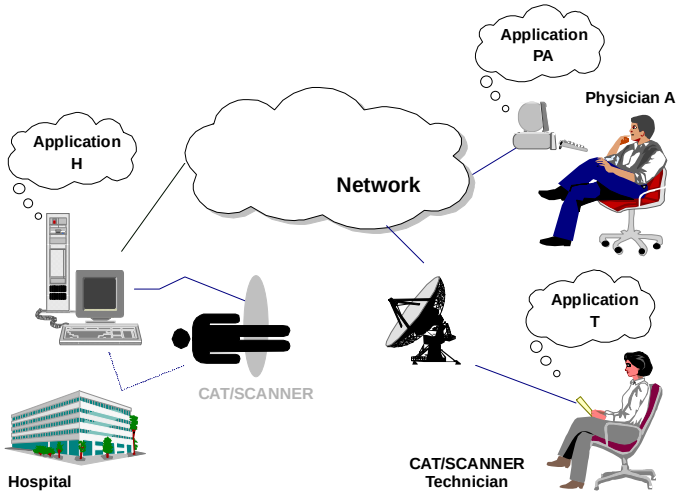

Figure 2.4: Remote Analysis of a CAT/scan with a CAT/scanner Technician

QuAL uses a string type name to identify a filter. Example 2.8 extends the declaration of the outport *image_output* (Example 2.1) of application H to include filtering (lines 4 through 9) on the multicast (line 10) of CAT/scan images. In the example, the multicast streams must be filtered because members of exam team differ on the amount of resources available for processing and displaying images. Thus, filtering is used to generate several traffic flow patterns from a single sender according to application specific criteria.

The ***realtm*** clause may include two sets of filters: (1) the filters that the port can comply with and (2) the filters that any other port must comply with to be bound to the port being declared. The keyword ***cmpl*** (line 4) identifies the first set, whereas ***conn_cmpl*** the second one. The binding mechanism in QuAL only binds ports that are able to comply

with each other's filter requirements. Filter negotiation will be further discussed in Section 2.5.

```
1 /* *** Application H (extension of Example 2.1) *** */
2 /* Specification of filters to control broadcast of CAT/scan images */
3 realtm {  loss ...        /* Specification of resource level QoS constraints */
4   cmpl {                  /* Keyword that declares the filters the */
5                           /* port is able to comply with */
6       low_qual;           /* List of filters the port is able to comply with */
7       med_qual;
8       high_qual;
9     };                    /* End of cmpl clause */
10    multicast;            /* Outport supports multicast */
11 }                        /* End of realtm clause */
12 receiveport {image_t} *image_output;
```

Example 2.8: Specification of Filters

A filter identifier in QuAL denotes communication QoS management in the same way a function name denotes the computation performed by a function. Identifiers are application dependent and have meaning only in the context of a particular port. In Example 2.8, the outport *image_output* can comply with the filters *low_qual*, *med_qual*, and *high_qual* (lines 6 through 8). In this context, they represent filtering policies that assure, respectively, transmission of one third, one half, and all images generated for the port. Section 2.5.2 explains how application T can select the filter *med_qual* for its communication with *image_input*.

Filters can be applied on a single communication stream or on a group of streams. *Single-stream filters* define constraints that are related only to a particular stream, such as the ones defined for *image_outport* in Example 2.8. *Multiple-stream filters* (or *group fil-*

ters) define management that depends on QoS properties of a group of streams such as reducing both an audio and a video stream rates. One can view a group filter as having a single inspector managing the aggregate flow on a group of streams.

```
1 /* *** Multimedia Conference Application (Sender Part) *** */
2 /* Specification of group filters */
3 enum {audio, video} audio_video;
4
5 /* Port through which audio will be broadcast */
6 realtm {  loss ...              /* Specification of resource level QoS constraints */
7   grp_cmpl {                   /* Keyword that declares the group filters */
8                                /* the port is able to comply with */
9     audio_video[audio];        /* List of contracts the port is able to comply with */
10    };                         /* End of grp_cmpl clause */
11    multicast;                 /* Outport supports multicast */
12 }                              /* End of realtm clause */
13 receiveport {audio_msg_t} *audio_output;
14
15 /* Port through which video will be broadcast */
16 realtm {  loss ...              /* Specification of resource level QoS constraints */
17   grp_cmpl {                   /* Keyword that declares the group filters */
18                                /* the port is able to comply with */
19     audio_video[video];        /* List of filters the port is able to comply with */
20   };                           /* End of grp_cmpl clause */
21   multicast;                   /* Outport supports multicast */
22 }                              /* End of realtm clause */
23 receiveport {video_msg_t} *video_output;
24 ...
```

Example 2.9: Specification of Group Filters

Example 2.9 illustrates the specification of group filters. The example shows a fragment of a multimedia conference application. The fragment includes the definition of the outports through which audio and video will be broadcast to other conference participants. The outports defined, *audio_output* (lines 6 through 13) and *video_output* (lines 16

through 23), are able to comply with the group filter *audio_video* (lines 9 and 19). The keyword ***grp_cmpl*** (lines 7 and 17) identifies the declaration of group filters. The filter *audio_video* manages rate reduction of audio and video messages.

Filter identifiers are enumerations where each element represents one of the ports that are interrelated. When a port is capable of complying with a filter, its declaration must identify which element of the enumeration that port represents. In the example, the enumeration *audio_video* contains *audio* and *video* (line 3). The outport *video_output* represents the video element, whereas *audio_output* represents the audio element. Their representations are specified by the *video* (line 19) and *audio* (line 9) words inside the brackets following the word *audio_video* in their declarations.

2.5.2 Filter Negotiation

The binding mechanism assures that ports are bound only when their types have compatible filters. Binding ports must be *compatible at the sending and at the arriving end.* Two ports are compatible at the sending end if there is a non-null intersection of the set of filters the outport is capable of complying with (specified after the keywords ***cmpl*** in Example 2.8 or ***grp_cmpl*** in Example 2.9) and the one the inport demands. Similarly, two ports are compatible at the arriving side if there is a non-null intersection between the filters the inport supports and the ones the outport demands.

Example 2.10 illustrates filter negotiation. It shows a fragment of Application T in Figure 2.4 that includes the declaration of the inport *med_image_input* (lines 5 through 11) which will receive the images sent by application H defined in Example 2.8. Due to restricted resources, application T can only handle half of the images generated by appli-

cation H. Therefore, the specification of *med_image_input* demands that an outport bound to it applies the filter *med_qual* (line 8) discussed in Section 2.5.1. The keyword ***conn_cmpl*** (line 6) identifies the list of filters a candidate connecting port must comply with. At binding time, QuAL runtime calculates the intersection set between the filters supported by *image_output* (*low_qual*, *med_qual*, and *high_qual*) and the ones demanded by *med_image_input* (*med_qual*). It decides that the ports have compatible types since the intersection set is *med_qual* and guarantees that *med_qual* will be enforced between *image_output* and *med_image_input*.

```
1 /* *** Application T *** */
2 /* Specification of filters that an outport sending CAT/scan images to application T */
3 /* must comply with */
4 /* Inport med_image_input will receive data sent by image_output in Example 2.8 */
5 realtm { loss ...          /* Specification of resource level QoS constraints */
6   conn_cmpl {              /* Keyword that declares filters that other ports */
7                            /* must comply with in order to connect to this one */
8       med_qual;            /* Filter that the other port must enforce */
9     }                      /* End of conn_cmpl clause */
10 }                         /* End of realtm clause */
11 receiveport {image_t} med_image_input;
```

Example 2.10: Negotiation of Filters

Example 2.11 illustrates the negotiation of group filters. The code shows the declaration of the inports *audio_input* (lines 7 through 13) and *video_input* (lines 16 through 22) that will receive, respectively, audio and video messages from the ports *audio_output* and *video_output* defined in Example 2.9. The keyword ***grp_conn_cmpl*** (lines 8 and 17) in a port declaration identifies the list of group filters a candidate connecting port must comply with. Similar to negotiation of single stream filters, QuAL runtime calculates the intersec-

tion set between the filters supported by *video_output* (*audio_video**[**video**]***) and the ones demanded by *video_input* (*audio_video**[**video**]***). QuAL runtime finds the intersection set to be non empty and assures that *audio_video* is enforced in the communication between *video_output* and *video_input*. The binding between *audio_output* and *audio_input* proceeds similarly. It is important to notice that the binding mechanism requires that *video_output* satisfies the filter *audio_video**[**video**]***, and not simply *audio_video*, in order to be bound to *video_input*.

```
1 /* *** Multimedia Conference Application (Receiver Part) *** */
2 /* Specification of group filters outports sending video and audio must comply with */
3 /* Inports audio_input and video_input receive from outports defined in Example 2.8 */
4 enum {audio, video} audio_video;
5
6 /* Port through which audio will be received */
7 realtm { loss ...              /* Specification of resource level QoS constraints */
8   grp_conn_cmpl {             /* Keyword that declares the group filters */
9                               /* another port must comply to connect to this one */
10      audio_video[audio];     /* Filter with which another port must comply */
11      };                      /* End of grp_conn_cmpl clause */
12    }                         /* End of realtm clause */
13  receiveport {audio_msg_t} audio_input;
14
15  /* Port through which video will be broadcast */
16  realtm { loss ...           /* Specification of resource level QoS constraints */
17    grp_conn_cmpl {           /* Keyword that declares the group filters */
18                              /* another port must comply to connect to this one */
19      audio_video[video];     /* Contract with which another port must comply */
20      };                      /* End of grp_conn_cmpl clause */
21    }                         /* End of realtm clause */
22  receiveport {video_msg_t} video_input;
23  ...
```

Example 2.11: Group Filter Negotiation

2.5.3 Implementing Filters

Application developers must bind to each filter a *filter function* that will actually implement the filtering. A filter function can be any function written in a subset of C (discussed in Chapter 3). The subset has the property of allowing computation of worst case execution time for scheduling purposes. Filter functions for outports (inports) are invoked every time a message is sent (received). Each filter function is executed on a separate thread created by the runtime when a connection is established.

Filter functions return values that indicate whether a message complies with given constraints. Filter functions return values of type ***mon_t*** in Definition 2.3. A positive value in the field *remove* indicates that the message must be filtered, whereas a positive value in the field *exception* indicates that an exception must be raised. Exception messages include the contents of the filtered message as well as the value of the *remove* and *exception* fields returned by the corresponding filter function.

Example 2.12 shows the definition of exception handler ports (lines 14 through 19) for the filters defined in Example 2.8, and Example 2.13 shows the corresponding filter functions (lines 27 through 54). Exception messages generated by the filters *low_qual*, *med_qual*, and *high_qual* are sent, respectively, to ports *appl* (line 16), *appm* (line 17), and *apph* (line 18).

Filter functions have no access to the contents of a message, only to its sending time, its arriving time, and an index that identifies the order of the message in the stream. Example 2.13 defines three filter functions: *mon_low* (lines 27 through 36) , *mon_med* (lines 37 through 46), and *mon_high* (lines 47 through 54). The first argument of a filter

function for an outport (lines 27, 37, and 47) is a message index and the second one its sending time. Filter functions for inports have an additional third argument that is the arriving time of a message. Appendix C describes signatures of filter functions.

```
1 /* *** Application H (Extension to Example 2.8) *** */
2 /* Specification of handler ports for filters */
3 realtm {  loss ...          /* Specification of resource level QoS constraints */
4   cmpl {                    /* Keyword that declares the filters the */
5                             /* port is able to comply with */
6       low_qual;             /* List of filters the port is able to comply with */
7       med_qual;
8       high_qual;
9     };                      /* End of cmpl clause */
10    multicast;
11 }                           /* End of realtm clause */
12 handlers {                  /* Keyword that declares handlers */
13                            /* for the port */
14    fil_handler{            /* Keyword that declares handler */
15                            /* for filters */
16      low_qual appl;        /* Handler port for filter low_qual */
17      med_qual appm;        /* Handler port for filter med_qual */
18      high_qual apph;       /* Handler port for filter high_qual */
19      }                     /* End of fil_handler clause */
20 }                           /* End of handlers clause */
21 receiveport {image_t} *image_output;
22
23 ...                        /* It continues in Example 2.13 */
```

Example 2.12: Implementing Filters (Part 1)

The filter functions in the example illustrate how to implement *media scaling*. Media scaling consists of sampling a message stream and transmitting only the fraction sampled. It assumes that the sampled data represents a good enough approximation of the original information. Media scaling can adjust the rate of messages in a stream according to the resources available. The function *mon_low* monitors the rate in which messages are gen-

erated at the outport and decides to uniformly drop messages to reduce the rate to one third. The function *mon_med* behaves similarly, but reducing the rate to only half. The function *mon_high*, on the other hand, does not drop any message. It just monitors the generation rate and raises exceptions whenever it is not 25 messages/s.

The ***assg*** operator associates filter identifiers to their corresponding functions. In Example 2.13, the functions *mon_low*, *mon_med*, and *mon_high* are associated to *low_qual* (line 59), *med_qual* (line 60), and *high_qual* (line 61), respectively. The ***assg*** operator enables applications to dynamically assign filter functions to filter identifiers.

2.6 Access to Communication Temporal Properties

For time sensitive applications, the semantics of the transmitted data depend on their *temporal properties*, i.e., the time when the data was sent, arrived, or processed. QuAL provides a set of functions to access such temporal properties.

QuAL runtime may prioritize message processing based on sending time. This is in addition to prioritization based on order of arrival, as supported by Concert/C. For example, this feature is useful for geology applications that process samples measured by remote seismic sensors. When an abnormal phenomena is detected, such as an earthquake, the analysis of data measured after the phenomena must have priority. All manipulations of temporal properties are performed by the QuAL runtime layer. This is analogous to network layer header processing being transparent to the transport layer.

```
22 /* Lines 1 through 21 in Example 2.12 */
23 #define RED_THIRD 1            /* Message removed to reduce rate to one third */
24 #define RED_HALF 2             /* Message removed to reduce rate to one half */
25 #define LOW_RATE 3             /* Exception caused by low rate */
26
27 mon_t mon_low (int index, double sending_time)
28 { /* Filter communication to provide low quality */
29   mon_t result;
30   if (index % 3 == 0)                 /* Transmit one message out of 3 */
31     result.remove = false;            /* Do not filter it; transmit it */
32   else
33     result.remove = RED_THIRD;        /* Filter message to reduce rate to one third */
34   result.exception = false;           /* No exception messages needed */
35   return(result);
36 }
37 mon_t mon_med (int index, double sending_time)
38 { /* Filter communication to provide a medium quality communication stream */
39   mon_t result;
40   if (index % 2 == 0)                 /* Transmit every other message */
41     result.remove = false;            /* Do not filter it; transmit it */
42   else
43     result.remove = RED_HALF;         /* Filter message to reduce rate to half */
44   result.exception = false;           /* No exception messages needed */
45   return(result);
46 }
47 mon_t mon_high (int index, double sending_time)
48 { /* Monitor communication to check the quality of the communication stream */
49   mon_t result;
50   result.remove = false;              /* Transmit every message unconditionally */
51   if(...)                             /* If rate is not 25 messages/s */
52     result.exception = LOW_RATE;      /* Generate an exception message */
53   return(result);
54 }
55 ...
56 main()
57 {
58   ...
59   assg(image_output, low_qual, mon_low);     /* Associate mon_low to low_qual */
60   assg(image_output, med_qual, mon_med);     /* Associate mon_med to med_qual */
61   assg(image_output, high_qual, mon_high);   /* Associate mon_high to high_qual */
62   ...
63 }
```

Example 2.13: Implementing Filters (Part 2)

Example 2.14 illustrates the retrieval of temporal properties in QuAL. The example shows the declaration of an inport, *image_input* (line 2), and a fragment of the message reception portion of an application. The ***receive_tm*** (lines 7 and 8) call, similarly to the Concert/C *receive* call, causes a message to be dequeued from *image_input* and stored in the memory position designated by *&m*. Additionally, ***receive_tm*** causes the sending, arriving, and processing times of the message retrieved to be stored in the memory positions designated by *&sendtm, &arrvtm, and &proctm,* respectively.

```
1   /* Access to Temporal Properties */
2   realtm { ... } receiveport {image_t} image_input;  /* Declaration of image_input */
3   ...
4   main()
5  {
6   ...
7   receive_tm(image_input, &m,          /* Retrieving an image frame and */
8        &sendtm, &arrvtm, &proctm);     /* its sending, arriving, and processing times */
9   ...
10  rtm_receive_tm(t, time(), image_input, &m,    /* Retrieving an image frame that */
11                                                /* arrived between t and time(), */
12       &sendt, &arrvt, &proct);                 /* and its temporal properties */
13  ...
14 }
```

Example 2.14: Access of Temporal Properties of Messages

The example also illustrates processing prioritization based on sending time. The call ***rtm_receive_tm*** (lines 10 through 12) dequeues from *image_input* only messages sent between *t* and the current time. The current time is retrieved by calling the function *time*. Function ***rtm_receive_tm*** also returns the temporal properties of the message retrieved.

---

typedef struct {int ***exception;*** *int* ***remove;*** *} mon_t****;***

Definition 2.3: Type of Value Returned by Filter Functions

Most language level communication abstractions [Hoare 78, Soares 92] do not capture communication temporal properties. Application developers must incorporate the sending time in the contents of the message with potential inaccuracies. Access to the arrival time is more complicated. The receiving process must check the time soon after a message is received, which may be difficult to guarantee in multi-process OSs due to potential scheduling problems. The time-stamping user process may be awaken only long after the message was received.

In QuAL, the manipulation of temporal properties is transparent to application developers. Furthermore, the time stamping process has high priority within the runtime with respect to other processes.

QuAL communication abstractions are compatible with the ones in Concert/C. All Concert/C operators can be used to access ports in which case the temporal properties are simply discarded.

2.7 Dynamic Re-negotiation of QoS Metrics

Re-negotiation of QoS constraints and filters in QuAL can happen dynamically, i.e., at any time during process execution and occurs concurrently with data transmission. This feature enables real time recovery from degradation without interrupting communication flow.

```
1 /* Dynamic Re-negotiation of QoS Constraints */
2 realtm {... delay ms 35; ...}              /* Declaration of image_input */
3   receiveport {image_t} image_input;
4 ...
5 main()
6 {
7   ...
8   /* A message was received with very high transmission delay ... */
9   qos_get (image_input)                    /* Check current delay for image_input */
10    {
11      delay sec &delay;
12    }
13    delay -= (ms) 5;                       /* Calculated new delay */
14    re_negotiate (image_input)             /* Start re-negotiation */
15    {
16      delay sec delay;                     /* Specify new value for transmission delay */
17    }
18    ...
19  }
```

Example 2.15: Dynamic Re-negotiation of QoS Constraints

Two operators enable dynamic QoS re-negotiation: ***qos_get*** allows retrieval of current QoS metrics of a port and ***re_negotiate*** causes the re-negotiation to start. Example 2.15 illustrates dynamic re-negotiation. The ***qos_get*** operator (lines 9 through 12) is used to access the value of the delay negotiated for the current connection serving *image_input* (returned in *&delay*). The new delay is calculated (line 13) based on the value retrieved (it is the current value minus 5 ms). Finally, the re-negotiation is requested through the ***re_negotiate*** operator (lines 14 through 17). QuAL runtime uses a two phase commit protocol [Elmasri and Navathe 94] to determine when the re-negotiation has terminated and the communication can proceed according to the new constraints.

2.8 QoS Management within the SNMP Framework

QuAL applications are automatically instrumented to generate, during execution, management information that captures the performance of the QoS delivered to them by the underlying system. The instrumentation is performed by the QuAL compiler and does not incur any implementation overhead to application developers. The information collected enables QoS management that focus on applications properties. Examples of such properties are average transmission delay delivered and deadlines to execute tasks. These informations are collected in QoS MIBs that are specified and maintained according to the SNMP standard.

The following sections provide a brief summary of the QoS MIB structure and focus on its use by applications. Chapter 4 provides a more in depth account on the design of QoS MIBs and their use by SNMP managers.

2.8.1 An Overview of the QoS MIB Design

A QoS MIB object is subdivided into the following groups (Figure 2.5):

- *Application* (qApp[2], for short): consists of the table qAppTable that contains one entry of type qAppEntry for each QuAL application. This group can be seen as an extension of the *Network Service Monitoring MIB* (*NSM MIB*) [Freed and Kille 93] to include information about application QoS.
- *Outport* (qOut): consists of the table qOutTable which has one row of type qOutEntry for each QuAL outport. Each entry indicates the value of all network

[2] 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 **Q**oS.

resource level QoS metrics (as defined in Section 2.3) negotiated for the outport and how the outport is using the connection.

- *Inport* (qIn): consists of the qInTable which has one row of type qInEntry for each QuAL inport. Similar to qOutTable, each entry in qInTable indicates the value of the metrics negotiated for an inport. In addition, it maintains some statistics on the QoS delivered by the network.
- *Programmable* (qProg): consists of the table qProgTable which has one row of type qProgEntry for each programmable (application specific) QoS metric. Entries are added to or removed from this group as applications trigger or cancel monitoring of new QoS metrics. Applications trigger QoS monitoring through the operator ***qual_monitor*** discussed in Section 2.4.1.

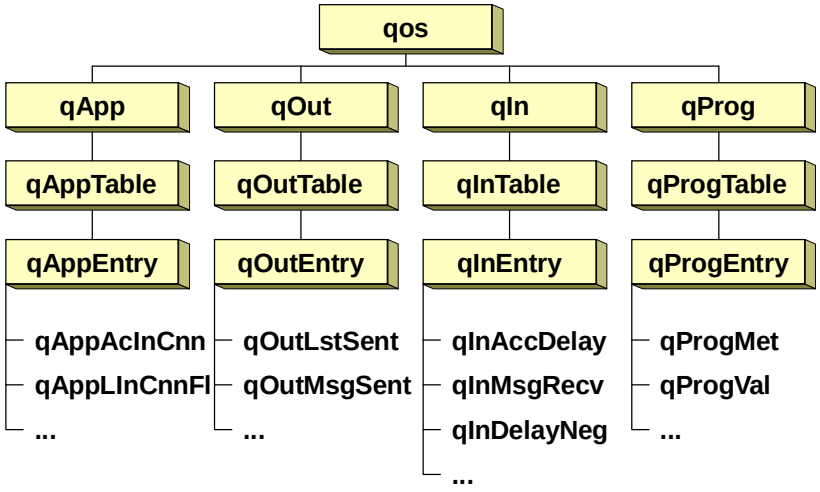

Figure 2.5: Overview of the Design of QoS MIBs

An entry in the application group captures the performance of its distributed activities and real-time executions. It contains information such as the application identifier, its current status (whether it is executing, blocked, or terminated), the last time when the OS

failed to schedule the application according to its timing constraints, the total number of its inports that succeed to connect (qAppAcInCnn in the figure), the last time when the application failed to establish a connection on an inport (qAppLInCnnFail), etc.

An entry in the outport group identifies an outport connection and the application that is using it. It describes the QoS negotiated and captures the performance actually being delivered. It contains information such as the identifier of the application that owns the outport, the maximum delay in milliseconds negotiated, the time when the last message was sent (qOutLstSent), the total number of messages sent (qOutMsgSent), etc.

Similar to an entry in the outport group, an entry in the inport group describes an in-port connection, the QoS negotiated, and the QoS being delivered. It contains information such as the sum of the transmission delay measured in milliseconds of all messages received (qInAccDelay), the total number of messages received (qInMsgRecv), the delay negotiated for the port (qInDelayNeg), the last time when a message was received, etc.

An entry in the programmable group contains information on a QoS metric defined by an application developer. Examples of such information are the name of the metric being measured (qProgMet) and its value (qProgVal).

2.8.2 Application Access to 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 accesses to MIB data. Once an application sends a request to an SNMP agent, there are no real-time guarantees on when the request will be served.

In order to provide real time QoS MIB access, QuAL includes functions that enable

applications to access the QoS MIB instrumentation directly, bypassing the local SNMP agents. From the application point of view, these functions have the same semantics as SNMP get [Rose 93, Stallings 93] operations. However, these functions offer real time response since they do not contend with an SNMP agent for processing resource. These functions are SNMP compatible because they comply with SNMP access control rules. That is, before accessing the instrumentation, they check whether or not the access complies with the SNMP specification of the QoS MIBs. QoS MIB access control functions have bounded computational cost.

Example 2.16 shows how application PA in Figure 2.1 accesses QoS performance statistics on the delivery of CAT/scan images. The example shows the declaration of the inport *image_input* (lines 3 and 4) that receives the CAT/scan images and fragments of the code that access QoS performance statistics associated with it (lines 6 through 22).

QoS MIB objects can be referenced by using SNMP identification conventions or QuAL constructs. The operator ***snmp_get*** is used to access QoS MIB data using SNMP conventions. It retrieves (lines 10 through 12) from the QoS MIB the number of messages already processed that arrived for *image_input*. The QoS MIB columnar object that contains this type of information is qInAccMsgProc. According to the design of QoS MIBs and the conventions established by SNMP, the instance of this object that stores the information for *image_input* is identified by the transport layer port address of *image_input* (1234, in the example), by the IP address of the machine where application PA is running (128.59.25.32), by the transport layer port address of the outport connected to *image_input* (4321), and by the IP address where the application communication with PA is running

(128.59.25.26). Application PA can acquire this information using various Unix libraries or using the operators to follow.

```
1 /* *** Application PA *** (Extended from Example 2.2) */
2 /* Accessing QoS performance statistics at an inport receiving CAT/scan images */
3 realtm {... delay ms 35; ...}              /* Inport that will receive CAT/scan images */
4   receiveport {image_t} image_input;
5 ...
6 main()
7 {
8   int msgProc, openConn, accDelay,
9   ...
10    /* Retrieve the number of messages received on image_input already processed */
11    msgProc =
12      snmp_get(qInAccMsgProc.1234.128.59.25.32.4321.128.59.25.26);
13    ...
14    /* Retrieve number of inport connections this application opened successfully */
15    openConn =
16      qual_app_get(self, qAppAccInCnn);
17    ...
18    /* Retrieve sum in ms of delay of all messages that arrived for image_input */
19    accDelay =
20      qual_in_get(image_input, qInAccDelay);
21    ...
22  }
```

Example 2.16: Real Time QoS MIB Access

The QuAL constructs differ from the abstractions in SNMP to identify QoS MIB objects. In QuAL, a port descriptor identifies a communication stream. In SNMP, however, lower level identifiers, such as transport layer addresses, are used to identify QoS MIB objects. This difference makes it very difficult for applications to identify the objects that store information on a QuAL 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.

To overcome cumbersome transport layer addresses, QuAL provides three operators to enable QoS MIB access using QuAL abstractions (one for each QoS MIB group). The operator ***qual_app_get*** identifies QoS MIB instances in the application group by using QuAL application abstractions. Similarly, the operators ***qual_in_get*** and ***qual_out_get*** use the inport and outport QuAL abstractions to identify, respectively, QoS MIB instances on the inport and outport QoS MIB groups. The call to ***qual_app_get*** (lines 14 through 16) retrieves the number of inbound connections application PA opened with success. The object that stores this type of information is qAppAccInConn, passed as the second argument. Its instance is the one associated with this application (***self***) identified by the first argument (the application invoking ***qual_app_get*** is always identified by ***self***). Similarly, the call to ***qual_in_get*** (lines 18 through 20) retrieves the sum in milliseconds of the transmission delays of all messages received. Object qInAccDelay stores this information and *image_input* identifies its instance. The result is returned in *accDelay*.

2.9 Conclusions

This chapter has described the QuAL language that enables applications to specify and manage (i.e., negotiate, monitor, analyze, and control) their communication and computation QoS. QoS management is pursued by applications and SNMP managers in coordination, based on QoS MIBs that contain effective statistics on the QoS delivered to applications by the underlying network and OS systems.

QuAL abstractions for QoS negotiation support multiple paradigms of interaction between applications and network QoS assurance mechanisms. At one extreme, applications can convey explicitly their QoS constraints to the network. At the other, applications

might include no constraints and expect the underlying system to adjust QoS delivery to match application behavior patterns. The QuAL runtime is responsible for bridging the gap between the paradigm chosen by applications and the one deployed by a particular network. Consequently, QuAL applications are portable across heterogeneous environments.

QuAL provides mechanisms for monitoring of QoS and for detection of QoS violations. Applications can specify the QoS to be monitored by the QuAL runtime which automatically stores measurements into QoS MIBs. The latter are structured and allocated as a result of the compilation process and can be queried by applications and SNMP managers. When violations are detected by the runtime, applications are informed by the QuAL runtime and can react accordingly.

QoS MIBs can be accessed by applications and SNMP managers to control QoS delivery. On the one hand, SNMP managers may monitor the QoS delivered to applications and operate transport and OS resources to best meet application needs. On the other hand, applications may evaluate their QoS performance and adapt their semantics accordingly. QoS MIBs integrate application customized QoS management into the standard SNMP network management framework.

Chapter 3

QoSockets: Unified Transport Interface for QoS Handling

3.1 Introduction

3.1.1 The Problem

Transport protocols very greatly in their support of QoS assurance. They range from no QoS support (such as in UDP), to limited support (such as in TCP), and up to intricate assistance (such as in ST-II). Even when providing considerable QoS support, they differ on the metrics that can be negotiated and on the negotiation mechanism supported. *The lack of transport homogeneity renders development of portable applications difficult in many ways:*

- *Application programmers need to be aware and handle the gap between the QoS needed and the one effectively supported by a particular provider.* For example, TCP does not support bandwidth allocation. Thus, a video application that needs to display 30 frames/s must explicitly allocate buffers.
- *Applications must handle details of QoS negotiation protocols.* For example, on one hand, ST-II sends asynchronous QoS negotiation status messages. On the

other hand, AAL sends synchronous QoS negotiation status information after it established connections. Applications need to incorporate all these negotiation mechanisms to be able to operate over ST-II and AAL.

- *Differences in the semantics of QoS assurance cause applications to work differently under different system configurations.* For example, AAL guarantees that transmission rate bounds are not violated whereas ST-II uses a best effort approach in delivering bandwidth. Thus, an application should deal with violations on ST-II but not on AAL.

Similar complexities in assuring QoS result from OS heterogeneity. For example, if the OS does not support real time computations, an application may miss the deadline to decode a video frame in time to display it. Such *application needs to know about OS performance behavior and react accordingly*.

The problem addressed in this chapter is that of designing unified transport layer API for QoS handling. It will also touch the problem of developing OS API for assuring QoS constraints, but only as far as supporting computations associated with transmissions over the transport API.

3.1.2 Main Results

This chapter presents QoSockets, a library of transport protocol API that abstracts and unifies QoS negotiation and management at the transport layer. It briefly overviews QoSOS, a similar library for OS functions. The functionality provided by QoSOS has been the subject of related research [Coulson et al. 95, Stankvic 95, Feldmeier 93, Govindan and Anderson 91] and thus this chapter focus primarily on QoSockets.

QoSockets add to existing communication APIs (such as Berkeley sockets [Stevens 90]) the ability to specify QoS constraints (e.g., delay or jitter) of a transport protocol. Similarly, QoSOS offers an API to negotiate QoS constraints with the underlying OS (e.g., deadlines to execute a task). Both QoSockets and QoSOS automatically monitor and gather QoS performance statistics.

The main contributions of QoSockets and QoSOS are:

- *Support of single API for transport layer QoS negotiation, connection establishment, and data transmission; and of single API for OS QoS negotiation.* This makes applications easy to port across different platforms because it hides the differences among QoS metrics supported by current transport protocols and OSs.
- *Support of a single QoS negotiation protocol.* QoS negotiation between peer applications is integrated with QoS negotiation between applications and the underlying system. Applications become part of the negotiation process according to each other's demands, regardless of the underlying transport or OS configurations. This feature hides underlying negotiation details from applications.
- *Generality across application QoS needs.* Applications can use the same API to request loss-less connections for voice communications or low bandwidth channel for electronic mail delivery. This feature eases implementation of diverse QoS needs and QoS upgrades or downgrades when porting to new environments.
- *Automatic monitoring of the QoS delivered by the underlying system and automatic detection of violations of QoS assurance.* This frees application developers from having to include tedious and error prone monitoring code.

- *Support of a flexible mechanism to dynamically select most appropriate QoS transport providers given specific application requirements.* This feature eases portability since the transports available are only know when migrating to new systems.

3.1.3 Chapter Organization

The reminder of this chapter is organized as follows. Section 3.2 describes how an application developer uses QoSockets to specify QoS constraints in a communication and how QoSockets allocate underlying system resources. Section 3.3 discusses how the QoSockets runtime hides from application developers the heterogeneity of transport protocols. Section 3.4 shows the QoSockets mechanisms that enable applications to dynamically select the most appropriate transport protocols. Section 3.5 discusses how QoSOS interacts with diverse OSs. Finally, Section 3.6 summarizes.

3.2 Specification of QoS Constraints in QoSockets

The Explicit Model in Chapter 2 suggests that applications should negotiate their QoS needs. QoSockets support such negotiation. In fact, the QuAL design uses QoSockets to implement its QoS negotiation portion. This section overviews QoSockets QoS specification.

Definition 3.1 specifies the *qos_ty* data type that enables the declaration of universal QoS metrics (as defined in Appendix A). For each universal metric, applications can specify a tolerable threshold value (field *value*), windows over which the metric should be measured (field *window*), and if the threshold can be coerced (field *coercion*) when bind-

ing with other sockets. QoSockets use the same coercion mechanism described in Section 2.3 of Chapter 2. The metrics in QoSockets are the same as the ones QuAL defines.

```
/* Definition of a QoS metric */
typedef struct qos_metric {
  int value          /* Tolerable threshold for QoS metric */
  int window;        /* How often (in seconds) the QoS metric must be measured */
  int coercion;      /* If the threshold 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 to the power -loss.value */
qos_met_ty permt;             /* A value higher than 0 in permt.value indicates */
                              /* that permutation is tolerated */
qos_met_ty min_rate;          /* Mean rate measured in messages/s must be */
                              /* higher than min_rate.value */
qos_met_ty rate;              /* Mean rate measured in messages/s */
                              /* must be lower than rate.value */
qos_met_ty peak;              /* Peak transmission rate is not higher than */
                              /* peak.value */
qos_met_ty delay;             /* End-to-end delay measured in ms must be */
                              /* lower than delay.value */
qos_met_ty jitter;            /* Jitter measured in ms must be lower */
                              /* than jitter.value */
qos_met_ty recovery;          /* Any recovery must take less than recovery.value */
int size;                     /* Maximum message size */
int multiple;                 /* Port supports a maximum of multiple connections */
                              /* concurrently */
int combined;                 /* QoS metrics measure the QoS on all */
                              /* connections combined */
} qos_ty;
```

Definition 3.1: Specification of QoS Metrics in QoSockets

Applications specify 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 *delay*

for port *p* specifies that the average delay cannot assume a value higher than 3 ms over intervals of 5 s on communications over *p*. The runtime uses *size*, when specified, to optimize resource allocation. Ports can 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 connection generates the *min_rate* and *rate* specified. The following section discusses how constraints can be associated to ports.

It is important to notice that QoSockets applications may also adhere to the Implicit and Intermediate Models in Chapter 2. The Implicit Model is implemented by omitting QoS specification for a port. A *NULL* value for a *qos_ty* object field indicates that the application chooses not to specify that particular constraint and leaves it up to the runtime. The Intermediate Model is supported by having the application provide QoS constraints and letting the QoSockets runtime choose the best service provided by the transport for the request.

Similar to QuAL, QoSockets runtime bridges the gap between the QoS assurance model (as in Chapter 2) chosen by applications and the one deployed by a network. In addition, it automatically monitors the execution of applications and dynamically re-negotiates QoS with the network to match application demands. Chapter 4 discusses how data collected during monitoring permits clever network management policies to adjust QoS delivery according to observed QoS behavior. For example, applications might choose the Implicit Model and networks might follow the Explicit Model. The QoSockets runtime will dynamically re-negotiate QoS with the network, according to application be-

havior.

QoSockets provide a single API for QoS specification that is independent of underlying transport mechanism details. The runtime translates abstract QoS specifications into transport specific service requests. Consider, for example, an application that chooses the Explicit Model. If the underlying system uses ST-II, QoSockets runtime must map abstract QoS rate and message size constraints into specific ST-II buffer size parameters. If the underlying system uses the Intermediate Model, the QoSockets runtime maps the application constraints into a service request for the class that best approximates application needs.

The semantics offered by QoSockets are that (1) it negotiates QoS with the network on a *best effort basis* and that (2) violations must be handled by applications. In a best effort QoS delivery, networks multiplex their resources in an effort to best fit the QoS requested without wasting resources. Stochastic models are used to characterize data traffic sources and predict when and where resources are needed. Nevertheless, there is no guarantee that violations will not occur during transient overload periods. As a consequence, QoSockets applications must be designed to handle violations and to adapt accordingly. When running on platforms that provide some support for QoS, however, these applications experience less violations than when running on platforms that make no such provisions.

3.3 QoSockets Connection Establishment Protocol

QoSockets *unify* several connection establishment protocols in one, promoting code portability and reuse. Figure 3.1 shows the time line for the typical communication sce-

nario using QoSockets and Figure 3.2 shows the QoSockets API system calls. In Figure 3.1, rectangles represent QoSockets function calls and the straight arrows represent execution flows. Time increases from top to bottom direction. Two execution flows are depicted: the Sender application and the Receiver application. The dashed arrows represent events handled by the QoSockets runtime concurrently with the execution of other tasks. These events are triggered by the system call where the arrow initiates. The balloon indicates when QoS negotiation happens.

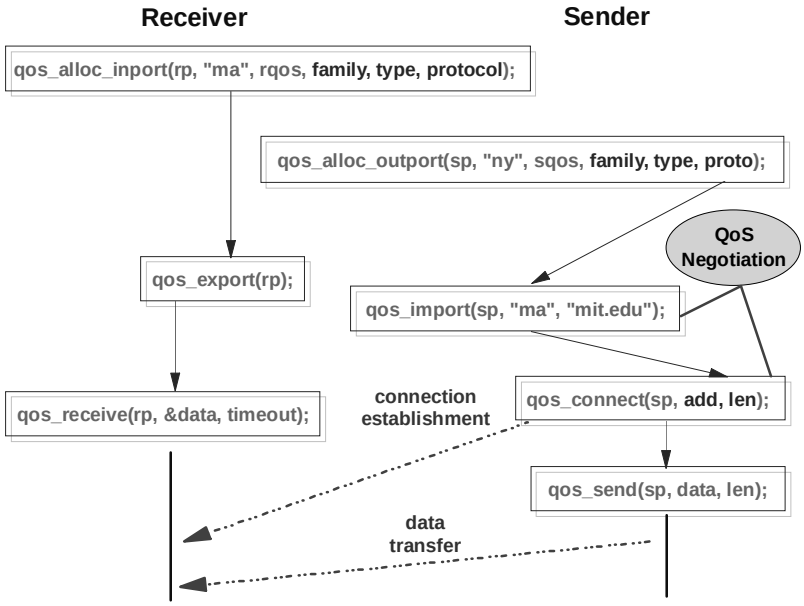

Figure 3.1: QoSockets Function Call Sequence to Establish a Communication

Ports in QoSockets can be identified by name (of type string) and do not need to be bound to a specific transport level port number. This feature increases code portability by preventing application failure due to conflicts on the allocation of transport level addresses. The name of a QoSockets port and its QoS requirements are defined at allocation time. Inports and outports are allocated, respectively, through the *qos_alloc_inport* and

qos_alloc_outport operators calls. In Figure 3.1, Receiver calls *qos_alloc_inport* to allocate inport *ma* (short for Massachusetts) with the QoS requirements expressed in the variable *rqos* of type *qos_ty* (Definition 3.1). At the end of the call, variable *rp* points to a descriptor for the allocated port. The last three arguments are optional and indicate the family (Unix internal protocol, Internet protocol, etc.), type (stream socket, raw socket, etc.), and protocol (if a specialized one like ICMP [Stevens 90], SPP [Stevens 90], etc., is needed). These arguments can have a *NULL* value in which case the QoSockets runtime automatically selects them based on the QoS requirements and on the protocols supported by the communicating machines. Consider, for example, an application running on a distributed environment that supports AAL and TCP. QoSockets runtime selects AAL when the application specifies QoS constraints and TCP when it does not.

qos_alloc_inport(inport_ty *port_ref, qos_ty qos_ref, int family, int type, int protocol);	Allocates an inport with the QoS constraints specified in *qos_ref* for transmission over a given communication *family*, *type*, and *protocol*. Returns in *port_ref* a reference to the inport created.
qos_alloc_outport(outport_ty *port_ref, qos_ty qos_ref, int family, int type, int protocol);	Allocates an outport with the QoS constraints specified in qos_ref for transmission over a given communication family, type, and protocol. Returns in port_ref a reference to the outport created.
qos_export(inport_ty *port_ref, char *external_name);	Publishes the QoS constraints and protocol specific addresses associated with *port_ref*. The information published is identified by *external_name*.
qos_import(outport_ty *port_ref, char *external_name, char *machine_name);	Connects port_ref to the port identified by external_name available on machine_name.
qos_connect(outport_ty *port_ref, struct sockaddr *addr, int addrlen);	Connects *port_ref* to the address specified in *addr*. *addrlen* has the size of the *addr* data structure.
qos_send(outport_ty *port_ref, char *data_ref, int len);	Sends *len* bytes of data stored in *data_ref* through *port_ref*.
qos_receive(inport_ty *port_ref, char *data_ref, struct timeval *timeout);	Blocks for a maximum of *timeout* waiting for data to arrive in *port_ref*. Saves in *data_ref* the first message that arrives before *timeout* expires.
qos_wait_inport_connected(inport port_ref, struct timeval *timeout);	Blocks for a maximum of *timeout* waiting for a connection to be established in *port_ref*.
qos_wait_outport_connected(outport port_ref, struct timeval *timeout);	Blocks for a maximum of timeout waiting for a connection to be established in port_ref.
qos_bind(port_ty *port_ref, struct sockaddr *addr, int addrlen);	Assigns the protocol level address specified in addr to port_ref. addrlen has the size of the addr data structure.

Figure 3.2: QoSockets API System Calls

The binding mechanism in QoSockets incorporates QoS negotiation between peer applications in the sockets mechanism. In Figure 5.1, the call to *qos_export* publishes to other QoSockets applications all the information associated with inport *rp*, such as its name and QoS requirements. QoSockets publish port related information by using name servers [Halsall 92] to store and access the information published. On the sender side, *qos_import* binds outport *sp* with the inport *ma* available on machine *mit.edu*. Operator *qos_import* first accesses the name server to retrieve information on a particular inport. It checks the QoS restrictions of *sp* with the ones retrieved from the name server and decides whether or not they are *compatible* (as discussed in Section 2.3 of Chapter 2). If they are, the ports are bound and connection can be established any time after that. Otherwise, *qos_import* returns an error and indicates why they are not compatible.

QoSockets support several alternatives for QoS negotiation between peer applications. If both choose to specify QoS constraints, QoSockets will coarse them to a compatible intermediate QoS. If only one specifies QoS constraints, QoSockets runtime assumes that the other side is able to comply with the constraints and allocates resources accordingly.

QoSockets establish connections as follows. Operator *qos_connect* triggers the connection establishment at the sending side. It blocks until connection establishment has been initiated. The last two optional arguments are used to identify the connecting inport by its physical address, when necessary. The transport service provider for the communication (if no transport was specified when the connecting ports were allocated) is allocated by *qos_connect*.

At the receiver side, QoSockets runtime frees applications from servicing connection

requests and connection establishment details. The QoSockets runtime process incoming requests, accepting or rejecting connections based on their QoS needs and on the QoS offered by the transport service provider chosen for the communication. At the sender side, QoSockets runtime manages connection establishment confirmations or rejections without exposing applications to such details.

In synchronous protocols [Comer 91], connection might have already been established by the time *qos_connect* returns. In asynchronous protocols [Topolcic 90], connection establishment has only been initiated when *qos_connect* returns.

Operators *qos_send* and *qos_receive* are for data transmission. Operator *qos_send* sends through *sp* a message that is up to *len* bytes long stored in *add*. Similarly, *qos_receive* blocks for up to *timeout* milliseconds waiting for a message to arrive for *rp*. If a message arrives within the time period specified, *qos_receive* retrieves it and stores it in the memory designated by *data*.

Operator *qos_send* and *qos_receive* block until a connection is fully established on the port transmitting data. Blocking may be avoided by using the operators *qos_wait_outport_connected* and *qos_wait_inport_connected* before the first call to *qos_send* and *qos_receive*, respectively. These operators block until connection has been fully established, but do not transmit any data.

3.4 Selection of Transport Protocols and Port Addresses

QoSockets permit the selection of specific transport protocols as well as port addresses. Applications may need to select a protocol because of compatibility issues. For example, they might need to use the TCP/IP stack to communicate with other modules

using the same protocol. They may also specify port numbers to publish their services where other applications expect them to be. For example, the SNMP standard specifies that agents communicate with managers through the UDP port 161.

A transport protocol is selected by specifying the last three arguments of the *qos_alloc_inport* and *qos_alloc_outport* operators. The selected transport protocol is used on any communication through the port. For example, an application selects ST-II by passing *PF_STIP*, *SOCK_RAW,* and *0* as the last three arguments. The connection establishment fails when connecting applications select different protocols at each end.

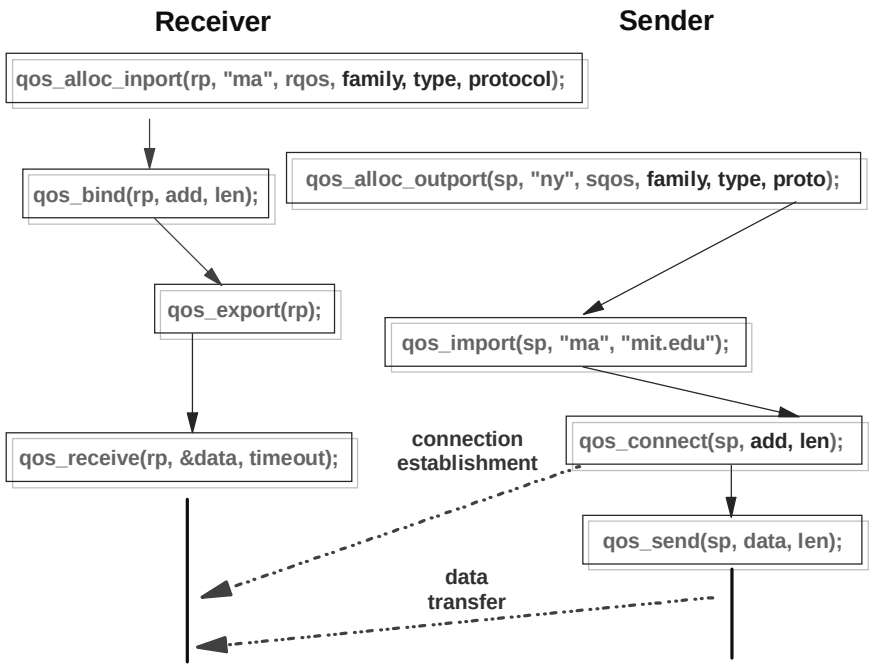

Figure 3.3: Identifying Inports by Names

Inports are bound to specific transport layer addresses by using *qos_bind.* After a physical address is selected, inports can be identified either by their names (Figure 3.3) or by their addresses (Figure 3.4). As Figure 3.4 illustrates, when identifying inports by ad-

dress, the call to *qos_import* is omitted. Only in this case, the call to *qos_connect* requires that the last two arguments be used to specify the address of the connecting inport.

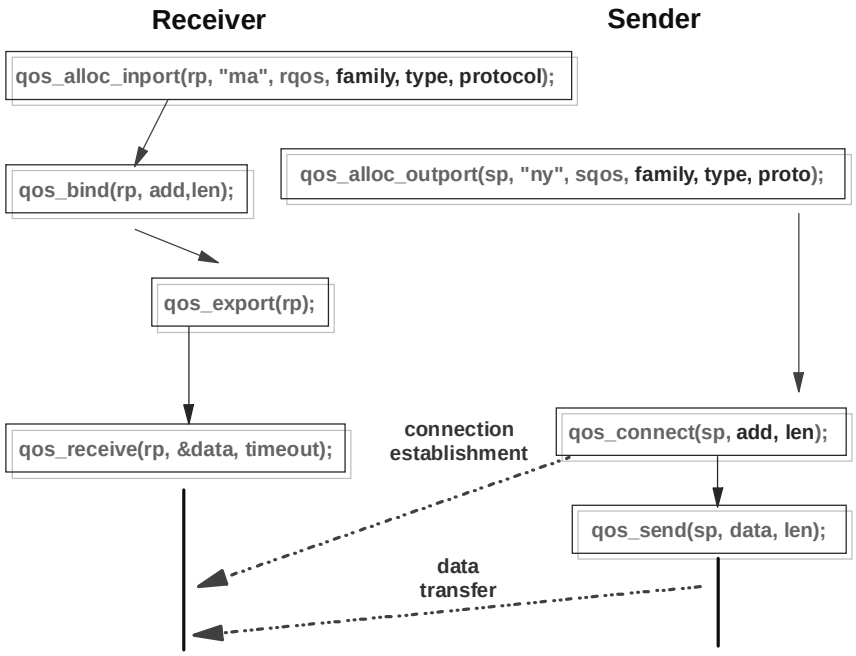

Figure 3.4: Identifying Inports by Transport Level Addresses

3.5 Handling Computing QoS Constraints

QoSOS manages processing resources in order to guarantee that applications are executed according to their computing QoS constraints. The main function of the QoSOS runtime is to bridge the gap between the QoS processing requirements of applications and the facilities provided by underlying OS. QoSOS can only guarantee real time constraints when the OS complies with the POSIX standard [Posix 90] which assures bounded execution time for OS kernel calls. Otherwise, QoSOS may monitor and notify violations, but not guarantee compliance to requested QoS. For example, some QoSOS applications need to be scheduled according to their deadlines. In this scenario, QoSOS guarantees that

applications are scheduled on an earliest deadline first basis, even if the underlying POSIX aware OS makes no provision for such scheduling mechanism.

In order to guarantee proper real time scheduling, QoSOS runtime must have preemptive execution priority with respect to other processes. One easy way to approximate such constraints is to run all user processes in the system using the QoSOS API. In this manner, the underlying OS will see only one user process running: the QoSOS runtime. This will be the assumption in the reminder of this chapter. Note that there are cases when such guarantees must be violated. For example, if a major power failure occurs, it may be necessary to backup the system immediately before the battery runs out. Such backup process has absolute high priority. In such rare extreme cases, the QoSOS runtime will miss its deadlines. Another example are system management processes such as accounting. These may shift the deadline, but can be estimated and minimized by running a single user process.

Additionally, QoSOS runtime must run during short and bound time intervals in order to avoid starvation of other important processes (such as device drivers or the scheduler). If this were not the case, for example, a network card could overflow its internal buffers and lose data.

QoSOS recognizes two modes of execution for applications: *real time mode* and *non real time mode.* Applications in the real time execution mode have to be scheduled according to their QoS computing constraints, while the ones in the non real time execution mode have no QoS constraints (and thus QoSOS makes no guarantees regarding when they will be executed).

The QoSOS runtime can be functionally divided into two main components: the *scheduleability analyzer* and the *real time scheduler*. The scheduleability analyzer manages the allocation of processing resources in the system. It keeps track of the capacity of the underlying processing system and the QoS constraints of each application executing in real time mode. Applications can only engage the real time execution mode if the scheduleability analyzer finds that there are enough processing resources available for the execution of the application. The real time scheduler is responsible for scheduling applications according to their QoS constraints. These two modules will be explained in Sections 3.5.1 and 3.5.2.

The QoSOS API provides the same functionality as the one provided by QuAL. It consists of a single operator, *qos_execute*, that is equivalent to the QuAL *within* block defined in Chapter 2. Initially, applications start execution in the non real time mode. They then invoke *qos_execute* to specify their QoS constraints in order to engage in the real time mode. A real time execution consists of the repetition of a task according to certain timing constraints.

To quickly review the definitions in Chapter 2, timing constraints cannot be missed in hard mode while they might be occasionally missed in soft mode. In the periodic case, tasks are to be executed repeatedly the number of times indicated by *period*. In the sporadic and aperiodic cases, the execution is triggered by *event*. In the sporadic case, however, there is a maximum number of times the event can happen, indicated by *period*.

Definition 3.2 shows the prototype of *qos_execute*. The argument *mode* indicates the real time execution mode. The argument *condition* indicates the condition to terminate the

real time execution. The argument *deadline_handler* is the handler invoked when soft constraints are missed. The last two arguments describe the ***do*** block of the real time execution.

```
/* Definition of the type of a block of instructions */
typedef void (*qos_fct)(void);

/* Definition of the execution mode type of applications */
typedef enum {
  soft_periodic;
  soft_aperiodic;
  soft_sporadic;
  hard_periodic;
  hard_sporadic;
  } qos_mode;

/* Definition of a timeout block */
typedef struct {
  int duration,          /* Maximum time period for the execution of the block */
  qos_fct action,        /* The sequence of instructions that constitutes the block */
  qos_fct handler,       /* Handler called when action takes longer */
                         /* than duration to execute */
  } qos_tm;

/* Prototype of the operator qos_execute */
int qos_execute (                  /* Returns the number of do_block executions */
      qos_mode mode,               /* Execution mode */
      int period,                  /* Maximum number of activations per second */
      qos_fct event,               /* Event that triggers executions */
      qos_fct condition,           /* Termination condition for real time execution */
      qos_fct deadline_handler,    /* Handler for execution deadline violations */
      int size,                    /* Number of timeout blocks of do_block */
      qos_tm **do_block)           /* Sequence of timeout blocks */
```

Definition 3.2: Definition of the Signature of qos_execute

As explained in Chapter 2, a ***do*** block contains ***timeout*** blocks and instructions with predictable computational cost. A ***timeout*** block consists of a sequence instructions (with

computational cost that cannot be calculated statically), a timeout period for its execution, and a handler that must be called if the execution does not finish before timeout expires.

For simplicity, *qos_execute* defines a ***do*** block as a sequence of only ***timeout*** blocks. Instructions with predictable computational cost are replaced by ***timeout*** blocks with *NULL* timeout. In Definition 3.2, the argument *size* indicates the number of ***timeout*** blocks that constitute the ***do*** block and *do_block* contains pointers to each one of these blocks.

The sixth argument indicates the number of timeout blocks (defined in Chapter 2) that constitute the *do* block (also in Chapter 2).

The operator *qos_execute* performs two main tasks. First, it interacts with the QoSOS scheduleability analyzer to check if the application can engage the real time execution mode. If the scheduleability analyzer returns a positive answer, then *qos_execute* notifies the real time scheduler of its timing constraints and waits to be scheduled accordingly. Otherwise, it finishes executing indicating to the calling application that it cannot engage real time mode at this moment.

The following sections will discuss the scheduleability analyzer and the real time scheduler in greater detail.

3.5.1 Scheduleability Analysis

QoSOS allocates resources for applications executing in hard real time mode based on the worst case execution performance analysis. A worst case performance analysis makes two pessimistic assumptions. First, it assumes that applications will always follow the most expensive execution path in their code. For example, consider an application that samples

a sensor placed in a volcano area. This application must sample data every interval of 1 s to guarantee that abnormal conditions are promptly detected. Depending on seismic conditions, the sensor can either provide 5 bytes of data (indicating a normal scenario) or 200 bytes of data (indicating an abnormal event). Even though in the majority of cases the sensor will provide only 5 bytes of data, the worst case performance analysis assumes that this application will always handle 200 bytes.

Second, the performance analysis considers that events that trigger sporadic applications always occur at the maximum possible rate. In this worst case scenario, sporadic applications behave like periodic applications. For example, consider the case of an application that handles alarms from devices in a distributed system. It must handle alarms as soon as they occur. Suppose that during major system failures the maximum rate is 15 alarms/s, but that otherwise alarms will seldom occur. In order to guarantee QoS, the scheduleability analyzer must allocate resources for 15 alarms/s.

The allocation of resources for applications executing in the soft real time mode is less restrictive than the one for the hard mode. QoSOS relaxes both pessimistic assumptions. It allows applications to engage real time execution mode even though they might have their QoS constraints eventually violated.

Traditionally, pure real time languages do not support constructs that can take arbitrarily long to execute, preventing the calculation of computational costs. In these languages, a task is required to satisfy the following requirements:

- *No recursion or cycles in the function call chain.* The system is unable to calculate an upper bound on the depth of recursions. This makes it impossible to determine

at compile time the worst execution time or the size of the heap needed to store all the activation records.

- *No hierarchical memory access.* Remote memory access over the network may take an unbounded amount of time due to underlying protocol issues or network failures.
- *No dynamic memory allocation.* The system cannot guarantee, during compile time, if there will be enough memory to execute the program or if a garbage collection may be necessary. These operations can take an unpredictable amount of time.
- *No blocking or event-driven statements.* The scheduler would have to reserve the CPU for the entire period in which the event may happen. Blocking statements include I/O, and event-driven statements include the arrival of messages to a port.
- *No contention for resources.* The system cannot calculate how long it will take for a shared resource to become available when multiple processes are trying to access it.
- *All loops must be bounded.* That is, it must be possible to compute how many times the loop will execute in the worst case. This feature allows a worst case execution time estimation for loops.

The QoSOS scheduleability analyzer adopts a different strategy to figure out application computational costs. To avoid having to restrict the C constructs that can be used in real time blocks, QoSOS supports a timeout mechanism similar to the one in QuAL. That is, if the compiler cannot calculate statically the worst case cost of an instruction or se-

quence of instructions, the application developer specifies the maximum tolerable execution time. The QoSOS runtime raises an exception whenever the execution does not end within the predicted time period. This mechanism is more flexible because computational costs can be specified finely, even at the granularity of a single language level instruction.

The QoSOS scheduleability analyzer manages computing resources for hard real time applications based on results described in [Jeffay 91] summarized in what follows. A *task* is a sequence of instructions that is invoked by each occurrence of a particular *event*. In QoSOS, a task is a *do_block*, specified as the last argument to *qos_execute*. An event is a stimulus generated by a process that is either external to the system (e.g., interrupt from a device) or internal to the system (e.g., clock ticks or the arrival of a message from another process). It is assumed that events are generated repeatedly with some maximum frequency; thus the time between successive invocations of a task will be of some minimal length. Each invocation of a task results in a single execution of the task at a time specified by a scheduling algorithm. Formally, a task is a pair *(c, p)* where *c* is the computational cost, and *p* is the *period*, that is, the minimal interval between invocations of the task. The clock ticks are discrete events and *c* and *p* are expressed as multiples of the interval between clock ticks. For periodic tasks, *p* specifies the constant interval between invocations while, for sporadic tasks, *p* specifies the minimum interval between invocations. Their results show that the scheduleability of a set of sporadic and periodic tasks can be efficiently determined. More precisely, given that the cost and the period of the tasks are known, the scheduleability can be determined in linear time and the system load can be bound.

3.5.2 Scheduling Applications According to Their Timing Constraints

The main goal of the QoSOS real time scheduler is to bridge the gap between the scheduling needed by applications and the one supported by the underlying OS. It schedules applications based on a slight variation of the *Earliest Deadline First* (*EDF*) algorithm [Liu 73], for executions in hard or soft mode. EDF chooses the task with the earliest deadline. Ties are broken arbitrarily. The invocation of a task with an earlier deadline may pre-empty the execution of another one with a later deadline. The scheduler uses EDF for both hard and soft blocks, but soft applications are pre-emptied by hard ones. Applications executing in non real time mode have the least priority, that is, only execute in the absence of ready hard or soft ones.

The universality of the EDF algorithm for scheduling sporadic and periodic tasks without preemption is shown in [Jeffay 91]. That is, if any non-preemptive algorithm can schedule a set of sporadic tasks, then the EDF algorithm also can. The non-preemptive scheduling is a more restrictive case of the preemptive scheduling, which is universal as a result. QoSOS uses the later version of EDF.

The QoSOS real time scheduler design follows the *split-level* scheduler architecture described in [Govindan and Anderson 91]. Split-level scheduling is a scheduler implementation technique that minimizes interactions between the scheduler and the processes being scheduled, while correctly prioritizing *Light-Weigh Processes* (*LWP*) [Sun Microsystems 94] of different *Heavy-Weigh Processes* (*HWP*) [Sun Microsystems 94]. Multiple LWPs inside a HWP share a *User-Level Scheduler* (*ULS*). The ULS monitors the execution of its LWP and interacts with a *Kernel-Level Scheduler* (*KLS*), that monitors all of

the ULSs. KLS and ULSs share information about the LWP with highest priority (that should execute next) using the Unix shared memory mechanism [Stevens 90]. The KLS schedules the corresponding ULS and sleeps until another LWP (and its ULS) gets priority. The ULS schedules the respective LWP and performs any context switching needed. This scheme is general enough to implement any priority and ordering among processes.

The QoSOS scheduler was designed to work on any OS that provides a real time service access interface in accordance with the POSIX standard. The standard guarantees, among other features, that the OS kernel calls have a bounded worst case execution time. Therefore, even though the real time scheduler is running on top of another OS, it can predict kernel response and better approximate real time requirements. This approach is less efficient than leaving scheduling to the OS. However, in most cases, available technology provides enough processing power to make it feasible. Additionally, this design choice allows the support for QoSOS applications on top of several heterogeneous general-purpose platforms.

3.6 Conclusions

This chapter presents the QoSockets and QoSOS APIs that promote code portability and reusability by sheltering heterogeneity in the QoS functions offered by several transport protocols and OSs. The main contributions of such approach are:

- A single API that is independent of transport layer and OS specifics. The same application can use services from several transport protocols without any modification.

- The runtime offers a single QoS negotiation mechanism which automatically bridges gaps among different transport protocol and OS providers.
- Upgrades to support new protocols or new OSs can be accomplished by extending the runtime with the interface to the new architecture components.
- The QoSockets runtime can automatically select the most appropriate transport given QoS requirements.
- The runtime can automatically monitor the QoS delivered with low overhead. The collected data may be accessed by other local applications as well as external SNMP managers.

QoSockets and QoSOS can be incorporated in languages or used as libraries. The QuAL runtime uses QoSockets and QoSOS to implement its functionality (in a sense, QuAL runtime is the first application to use QoSockets and QoSOS). Other languages may be implemented in the same fashion. Finally, applications may use QoSockets and QoSOS directly as a C library.

Chapter 4

Managing QoS Delivery

4.1 Introduction

4.1.1 The Problem

The network management and the application QoS adaptation strategies will accomplish better results through coordination. Network management may improve the QoS in application streams by allocating alternative routes. Applications may operate under QoS degradation by adapting their streams to the QoS received. But, without coordination, these activities may settle at unsatisfactory or unstable operational points. For example, upon congestion at a switch, SNMP managers may decide to allocate alternative routes and, concurrently, applications may reduce their transmission rates. *Both* applications and managers need to understand requested and delivered QoS to coordinate their efforts.

The main challenges in providing this coordination are:

- *How can QoSME disclose application level QoS performance to underlying system managers without incurring extra application development overhead?* The goal is to automate as much as possible the process of providing application QoS profiles.

- *What is the information that underlying system managers need to manage QoS performance according to application needs?* The goal is to find a small information set that will provide a picture of the application QoS behavior.
- *How can coordination between underlying system managers and applications happen?* The goal is to effectively avoid overlapping of corrective actions from applications and underlying system managers.

This chapter addresses these challenges.

4.1.2 Main Results

The main contribution of this chapter is an architecture for integrating application level QoS management and underlying system management. The architecture proposed consists of QoS MIBs and SNMP agents that provide QoS MIB access to SNMP managers. The objects in these MIBs deal with application level information, such as video frame delays and voice stream jitter. The information is partitioned among MIB groups according to applications, outports, inports, and programmable (application specific) metrics. The proposed architecture has the main novel advantages:

- *It includes a mechanism to disclose application level QoS performance to underlying system managers.* By accessing QoS MIBs, managers of transport layer connections can identify, for example, the application that is using a particular connection, the QoS the application requested, and the QoS that is being delivered to it. This information may be used to decide alternative allocation policies or to pin point applications that are overloading network resources.

- *It contains the necessary information to characterize application QoS behavior.* QoS MIBs store the QoS requested by applications and measurements on the QoS being delivered to them. These measurements include the value of universal metrics (such as end-end delay, jitter, loss, etc.) and application specific ones, as defined in Chapter 2.
- *It includes coordination of application and SNMP management activities.* This is useful when applications and managers detect violations and try to compensate for them. They should coordinate their activities to avoid interfering with each other's decisions. For example, an SNMP manager may decide to allocate more throughput in some network links to overcome congestion while applications may decide not to decrease their transmission rates because they are aware of this management decision.

4.1.3 Chapter Organization

The reminder of this chapter is organized as follows. Section 4.2 gives an overview of QoS management through QoS MIBs. Section 4.3 overviews the design of QoS MIBs. Sections 4.4, 4.5, 4.6, and 4.7 discuss the QoS MIB data stored per application, per outport, per inport, and per programmable metric, respectively. Section 4.8 presents the challenges in instrumenting QoS MIBs. Finally, Section 4.9 summarizes.

4.2 Overview of the QoS Management Architecture

Figure 4.1 illustrates the architecture for QoS management through QoS MIBs using a generic multimedia multi-application example. The applications sample input devices (such

as monitors, cameras, and microphones), broadcast them to other participants, and finally display received samples locally. The runtime instances at each site support interactions between applications and the underlying transport and OS, and store in QoS MIBs information on the QoS effectively received. Examples of such information are the amount of bandwidth allocated and received in the communication. SNMP agents embedded in the architecture provide QoS MIB access to SNMP managers.

Applications read QoS MIB fields to detect QoS violations and update them to trigger corrective actions. Consider, for example, an inport receiving video data that requires a delay not higher than 5 ms over windows of 1 s. The QuAL runtime type checking mechanism automatically monitors delay variations and signals the application when a violation occurs. However, video play-out time may also require adjustment when the average delay is lower than a certain threshold (for example, 3 ms). Low average delays may not violate the type of a port and therefore may not be automatically detected by the runtime. Applications need to query QoS MIB objects to detect such situations.

SNMP managers use QoS MIB data to manage QoS delivery based on application needs. These managers are exposed to the configuration of applications running on the system and can customize their service management accordingly. For example, when the delay on a communication is higher than the application expected, a manager[1] can initiate the establishment of an alternative connection. Managers can also use information on other SNMP MIBs to aid the analysis and control of QoS violations. For example, by monitor-

[1] If many SNMP managers try to update the same MIB object concurrently, the SNMP standard does not guarantee transaction atomicity. Atomicity can be accommodated employing protocols that use special MIB objects as semaphores and forcing all managers to adhere to the protocol. This is, nevertheless, a limitation of the SNMP model and not of the QoS MIB design.

ing ATM switch MIBs, a manager can force communication establishment to bypass congested switches. The QuAL runtime includes an SNMP agent that provides QoS MIB access to SNMP managers.

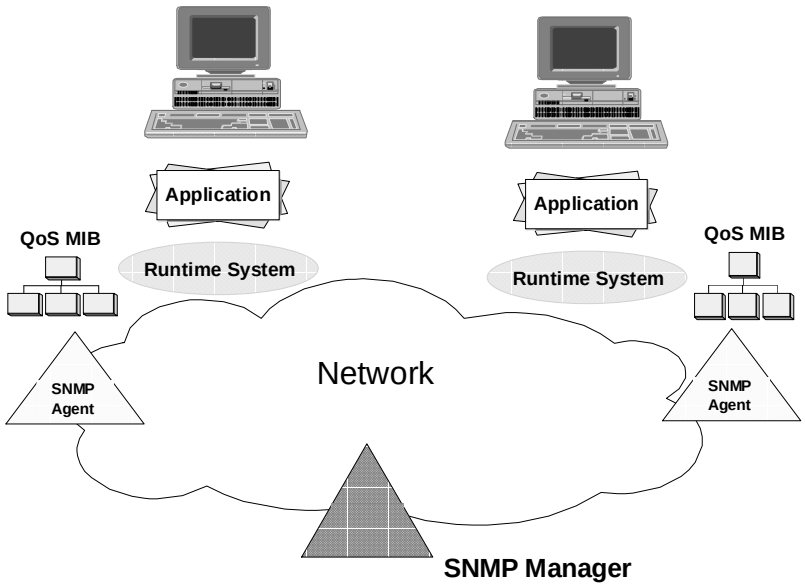

Figure 4.1: Overall Architecture for Instrumentation and Access of QoS MIBs

Figure 4.2 depicts in more details the interactions between the runtimes and the SNMP entities. The lenses represent the instrumentation for QoS MIB data collection, the full lines represent data flow, and the dashed arrows show management information flow. Instrumentation is added at the interface between applications and the runtime components that provide QoS demanding services. In QoSME, these components are QoSOS and QoSockets. As discussed in Chapter 3, QoSOS schedules applications according to their real time constraints, and QoSockets mitigate the flow of multimedia application streams across network resources. It is important to notice that the main concepts of using QoS MIBs and their automatic instrumentation to manage QoS is independent of QoSME and can be implemented in other environments.

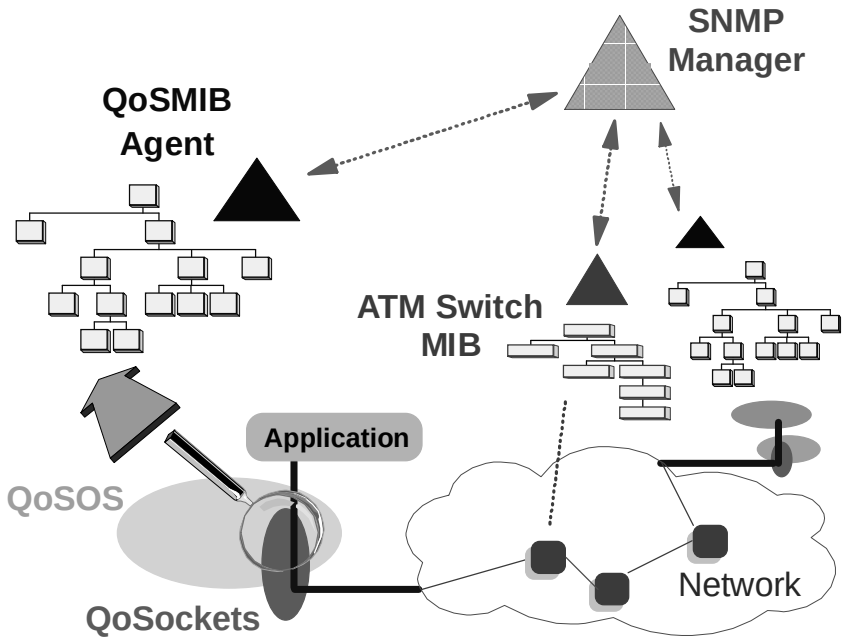

Figure 4.2: Architecture for Instrumentation and Access of QoS MIBs

The instrumentation snoops all QoS demanding interactions between applications and runtime systems to collect management information, such as the last time when a protocol stack established a connection for an application. The information collected by the instrumentation is stored in QoS MIBs. The overhead of such instrumentation is discussed in Chapter 5.

SNMP managers use information on other SNMP MIBs to aid the analysis of QoS violations. Figure 4.2 depicts an SNMP manager that also monitors an ATM switch MIB values to understand the QoS in communications between applications. The SNMP manager traces, for example, cases where unexpected transmission delays are caused by a congestion in the switch. In such scenario, managers can automatically request the establishment of an alternative connection that bypasses the congested switch.

To understand how the several modules depicted in Figure 4.2 interact, consider, for

example, the management of voice transmission quality. QuAL compilers automatically generates instrumentation at the appropriate interfaces when applications use QoS constructs (in QuAL), or QoSOS and QoSockets (in C). Recall that QuAL QoS constructs are implemented using QoSockets or QoSOS and thus it suffices to consider how the latter operates. QoSockets stores in the QoS MIBs the time when a voice connection was requested, the time when it was established, the QoS negotiated with network (such as the mean transmission delay and the mean rate), and the actual transmission delay experienced by voice samples. QoSockets computes the latter information by using the time stamps it adds to messages. Section 4.8 discusses the instrumentation of QoS MIBs in greater detail.

QuAL provides constructs that provide direct instrumentation access to applications, bypassing SNMP agents. These constructs were discussed in Section 2.8 of Chapter 2. They check if the invoking application has the appropriate access rights for the requested information, verify if the local instrumentation is indeed collecting the requested information, and access the instrumentation directly. Such mechanism lowers the response time to local MIB accesses without violating SNMP access constraints.

Other QuAL constructs can be used to handle QoS violations. For example, voice applications need to quickly respond to degradation in the transmission quality. One possible approach is to adjust the play out time of voice samples according to the mean jitter of the communication. A mean jitter higher than expected may be overcome temporarily by smoothly delaying the play out time of voice packets.

Corrective measures can be initiated by an SNMP manager that oversees the voice

quality. The manager interacts periodically with the SNMP agents to retrieve the mean jitter of the voice communications. Upon detection of poor transmission quality, it initiates QoS performance control by re-configuring the allocation of network resources. In this scenario, the response time of the manager control actions depends on the frequency at which the manager is polling the agent and on the delay incurred by SNMP to access MIB values. These issues are inherent in the SNMP design. Reference [Goldszmidt 95] discusses alternative designs for SNMP management applications that overcome these limitations.

QoS MIBs integrate application management within general network management frameworks. This feature is important because it may become inefficient or intractable to manage distributed application activities using only lower layer information. This difficulty comes from the increasing gap between application level abstractions and underlying system entities providing services.

The architecture presented provides a framework for dividing management responsibilities between SNMP managers and applications. On one hand, authorized SNMP managers can access QoS MIBs and manage the underlying system according to application needs. On the other hand, applications can manage themselves, according to the received QoS.

4.3 An Overview of the QoS MIB Design

The QoS MIB data belongs to one of the following groups, as illustrated in Figure 4.3.

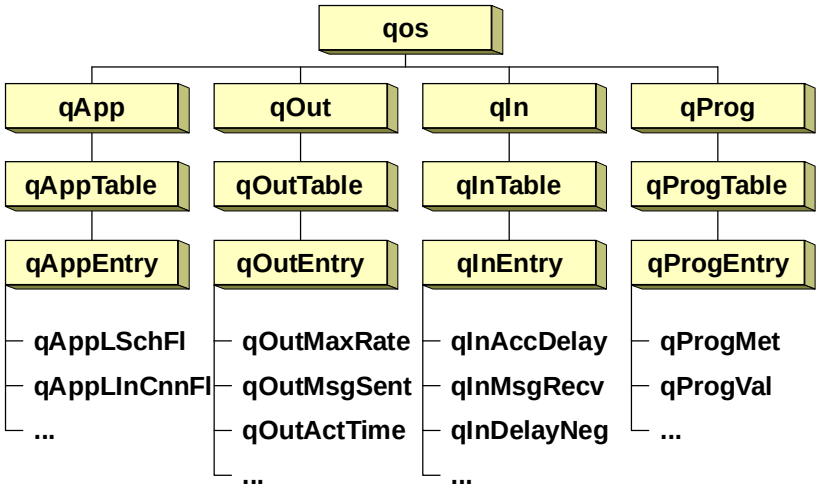

Figure 4.3: QoS MIB Object Groups

- *Application* (qApp[2] for short): Consists of the table qAppTable that contains one entry of type qAppEntry for each application running on the system. Each entry indicates the QoS provided by the underlying OS and general information on the response of the protocol stack to QoS demanding connection establishment requests. For example, the application group object qAppLSchFl stores the last time when the OS failed to schedule an application according to its timing constraints and qAppLInCnnFl stores the last time when an application had a connection establishment request rejected. This group can be seen as an extension of the NSM MIB to add information about application QoS.
- *Outport* (qOut): Consists of the table qOutTable which has one row of type qOutEntry for each outport connection of applications in qApp. Each entry indicates the QoS negotiated for the outport and how the outport is using the connec-

[2] 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 **Q**oS.

tion. For example, the object qOutMaxRate indicates the rate negotiated[3] with the network at connection establishment time, qOutMsgSent indicates how many messages have been sent so far, and qOutActTime indicates when the connection became active. A manager[4] can calculate the average transmission rate effectively received by dividing qOutMsgSent by the time elapsed since qOutActTime. It can then compare the result with qOutMaxRate which holds the negotiated rate.

- *Inport* (qIn): Consists of the table qInTable which has one row of type qInEntry for each inport of applications in qApp, similar to qOutEntry. In addition, it maintains measures on the QoS effectively delivered by the network. For example, the qInDelayNeg, qInAccDelay, and qInMsgRecv objects indicate the transmission delay negotiated with the network, the sum of the transmission delays of all messages received, and the number of messages received, respectively. An SNMP manager uses these data to establish alternative paths for connections that are experiencing a mean transmission delay much higher than the one negotiated. The mean transmission delay is calculated by dividing qInAccDelay by qInMsgRecv.
- *Programmable* (qProg): Consists of the table qProgTable which has one row of type qProgEntry for each application programmed (application specific) QoS metric. An entry in qProgTable indicates the metric that is being measured, the application that requested the measurement, and the last value measured. For example, the objects qProgMet identifies a metric and the object qProgVal indicates the last

[3] In QoSME, QoS can be negotiated either through QuAL abstractions (Chapter 2) or through QoSockets API (Chapter 3).

[4] The term manager refers to *any* application managing QoS performance. It may be an SNMP manager controlling underlying system resources or a QuAL application adapting to QoS delivery.

value measured. Entries in this group are added or removed as applications trigger or cancel monitoring of new QoS metrics, as discussed in Chapter 2. Metrics that use the window mechanism to specify the measurement frequency (discussed in Chapter 2) are stored in this table.

The information stored by the columnar objects of the MIBs presented in the following sections will be classified in one of the following categories:

- *Identification*: used to describe a particular instance of an object;
- *Configuration*: used to identify how resources were allocated for a service;
- *Operational behavior statistics*: used to analyze the actual performance delivered by the underlying system, and
- *Coordination*: used to synchronize management actions between applications and SNMP managers.

4.4 QoS MIB Data per Application

The application group stores QoS performance statistics of application real time computations, as illustrated in Table 4.1. Object 1 is of type identification, objects 2 through 4 are of type configuration, objects 5 through 19 are of type operational behavior statistics, and objects 20 and 21 are of type coordination. The '*' in the object qAppId indicates that it is an *index* object. Its instances uniquely identifies an entry in the qAppTable. Application group objects also indicate the response time of the underlying transport to QoS demanding connection establishment requests, as illustrated in Table 4.2. All objects in this table store statistics.

#	Object	Syntax	Description
01	qAppId*	OBJECT IDENTI-FIER	Process identification
02	qAppOperStatus	“nrt” | “hp” | “hs” | “sp” | “ss” | “sa” | “down”	The operational status of the application, that can be executing in the non real time mode (nrt), or in real time. In the last case, the first letter indicates the mode (hard or soft), and the second letter the behavior (periodic, sporadic, or aperiodic). The value “down” indicates that the application has terminated.
03	qAppPeriod	INTEGER	Number of times per second that the real time computation should be scheduled
04	qAppCost	INTEGER	Estimated computational cost in milliseconds of the real time computation in execution
05	qAppLstChng	TimeStamp[5]	Time when the application entered current state
06	qAppUpTm	TimeStamp	Time when the application started
07	qAppAccSoft	Counter32	Total number of times the application executed in soft real time mode
08	qAppAccHard	Counter32	Total number of times the application executed in hard real time mode
09	qAppAccSoftTm	INTEGER	Total amount of time the application executed in soft real time mode
10	qAppAccHardTm	INTEGER	Total amount of time the application executed in hard real time mode
11	qAppLstSoft	TimeStamp	Last time when the application executed in soft real time mode
12	qAppLstHard	TimeStamp	Last time when the application executed in hard real time mode
13	qAppMissDead	Counter32	Number of times the application missed a deadline
14	qAppLstSchdFail	TimeStamp	Last time when a deadline was missed
16	qAppExpTmout	Counter32	Number of times the timeout for executing a real time task expired
17	qAppLstExpTmout	TimeStamp	Last time when a timeout expired
18	qAppLstSoftFail	TimeStamp	Last time when a request to execute in soft real time mode was rejected due to lack of processing resources available
19	qAppLstHardFail	TimeStamp	Last time when a request to execute in hard real time mode was rejected due to lack of processing resources available
20	qAppManager	DisplayString	Entity currently managing QoS violations
21	qAppMgtStatus	DisplayString	QoS violation control request from an application to an SNMP manager or vice versa

Table 4.1: Application Group Objects for Real Time Computations

A local SNMP manager functioning as a scheduler can use application group objects to detect whether application performance degradation is caused by inadequate OS schedul-

[5] 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 management system, than the respective TimeStamp object contains a zero value.

ing policies or 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 broken and the application has subsequent connection establishment requests rejected. An SNMP manager can use qAppLstSchdFail and qAppLstInCnnFail to retrieve the type of the last failure. If the OS failed to schedule the application on time, the scheduler might decide to change the allocation of processing resources. If connections cannot be established, it might decide to notify the problem to another manager capable of handling the problem. At the same time, applications can use qAppLstSchdFail and qAppMissDead to monitor the level of QoS they obtain from the OS. If the OS consistently fails to meet scheduling deadlines, applications can trigger requests to allocate more processing resources.

#	Object	Syntax	Description
01	qAppLstInCnnFail	TimeStamp	Time when the last connection to a QoS demanding inport was rejected
02	qAppLstOutCnnFail	TimeStamp	Time when the last connection to a QoS demanding outport was rejected
03	qAppLstInCnn	TimeStamp	Time when the last connection to a QoS demanding inport was established
04	qAppLstOutCnn	TimeStamp	Time when the last connection to a QoS demanding outport was established
05	qAppInCnn	Counter32	Total number of open connections to QoS demanding inports
06	qAppOutCnn	Counter32	Total number of open connections to QoS demanding outports

Table 4.2: Application Group Objects for Communication Activities

The application group includes coordination control objects between applications and SNMP managers. 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 in isolation. For example, an SNMP manager responsible for load balancing updates qAppManager to indicate load re-distribution. During load balancing, applications cannot change their deadline constraints.

SNMP managers responsible for scheduling use configuration data such as qAppPeriod and qAppCost to manage allocation of processing resources. In situations where several deadlines are missed, an SNMP manager may choose to request applications to relax their timing constraints. In such case, qAppManager indicates that an SNMP manager is not currently controlling QoS violations and qAppMgtStatus informs the request from an SNMP manager to the application to loosen its real time constraints.

4.5 QoS MIB Data per Outport

The goal of the outport group is to inform about outport connections, their QoS requirements, how they are being utilized, and to coordinate management of their QoS performance between applications and SNMP managers. This group also includes information on connection problems and recovery performance.

Identification objects store the local and remote addresses of the communicating machines, identifiers 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.

The following sections discuss in greater detail the configuration and operational be-

havior of outport group objects.

4.5.1 Configuration Outport Group Objects

Table 4.3 shows the configuration objects present in the outport group. SNMP managers use configuration objects to guide the allocation of communication resources per outport connection. The qOutMsgSize object indicates the maximum size of messages transmitted on a connection. The qOutProtocol object identifies the transport protocol serving the connection. The qOutLoss, qOutPermut, qOutMinRate, qOutMaxRate, qOutPeak, qOutDelay, qOutJitter, and qOutRecTime objects identify the QoS constraints negotiated for the outport, as discussed in Chapters 2 and 3. These data enable an accurate analysis of the resources allocated per connection.

#	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 number of messages per second
05	qOutMaxRate	INTEGER	Maximum number of messages per second
06	qOutPeak	INTEGER	Peak number of messages per second
07	qOutDelay	INTEGER	Maximum propagation delay
08	qOutJitter	INTEGER	Maximum jitter
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 4.3: Configuration Outport Group Objects

Consider, for example, an application that receives radiology images and occupies most of the communication resources on a machine. If other applications are unable to open connections, a local SNMP manager can use qOutMaxRate, qOutPeak, and qOutMsgSize object instances to calculate how buffering resources are currently distrib-

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

The qOutManager object enables management coordination between applications and SNMP managers. 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. The object qOutManager will indicate that the application is controlling the loss rate violation, inhibiting other SNMP managers from initiating any control action such as finding alternative paths for the communication.

4.5.2 Operational Behavior Statistics Outport Group Objects

Table 4.4 illustrates operational behavior outport group objects. QoS managers use qOutCnnFail and qOutAccRecTime object instances to estimate recovery time from connection failures and manage QoS performance accordingly. For example, the average recovery time can be calculated by dividing qOutAccRecTime by qOutCnnFail. Thus, an application unable to send data over a connection due to a failure can decide whether to open an alternative connection or to wait for recovery based on the mean recovery time.

SNMP managers use operational behavior objects, such as qOutMsgSent, and qOutVolume, to evaluate how much of the resources allocated by an application are actually

being used, and to re-negotiate QoS if the utilization ratio is low. An SNMP manager reduce a communication allocation from 30 frame/s video to 15 frame/s if the application has not sent more than 15 frames/s recently. By detecting under-utilization, managers can allocate resources more efficiently.

#	Object	Syntax	Description
01	qOutCnnFail	Counter32	Total number of connection failures
02	qOutAccRecTime	INTEGER	Total amount of time spent in recovering
03	qOutEstTime	TimeStamp	Time when the connection was established
04	qOutActTime	TimeStamp	Time when the traffic became active
05	qOutMsgSent	Counter32	Total number of messages sent
06	qOutVolume	Counter32	Total volume of data sent in kilobytes
07	qOutLstMsg	TimeStamp	Time when the last message was sent through the connection
08	qOutLstFail	TimeStamp	Time when last connection problem occurred
09	qOutStatus	“up” | “down”	Status of the connection

Table 4.4: Operational Behavior Statistics Outport Group Objects

4.6 QoS MIB Data per Inport

The inport group contains identification and configuration objects similar to the ones in the outport group objects and operational behavior statistics on the QoS delivered by transport service providers. These statistics are stored on qInTables because they can be calculated more efficiently on the receiving side than on the sending side of the communication.

A message arrives *out of sequence* when its order in the arriving stream is not the same as in the sending stream or it is considered lost, that is, its transmission delay is higher than the timeout (as discussed in Appendix A). In order to account for losses and permutations, separate statistics are maintained for messages that arrive *out of sequence* and those that arrive *in sequence*.

Messages that arrive out of sequence may or not consist a violation, depending on the application semantics. Some applications (e.g., video transmission) discard messages that arrive out of sequence, whereas others (e.g., management applications that perform commutative operations on data samples) use messages that arrive out of sequence. By partitioning the information, analysis can be performed on each flow in an application dependent manner.

Inport group objects also store operational behavior statistics on the *processed messages*, enabling the study of how applications are processing the data received. Processed messages are the ones received and processed by the respective application. For example, possible bottlenecks are identified when the rate at which messages are processed by an application is considerably lower than the one at which messages arrive. An SNMP manager can analyze the QoS delivered by the OS and verify if the application has been scheduled according to its execution deadlines. If so, the SNMP manager can request the application to adjust its timing constraints to be scheduled more often and increase the message processing rate.

Table 4.5 illustrates the operational behavior statistics objects that capture information on the traffic of messages that arrive in sequence. Similar sets of objects capture information on the traffic of messages that arrive out of sequence and on the traffic of processed messages.

Inport MIB operational behavior statistics objects enable the analysis of the universal QoS actually delivered to an application. The *delay* of messages that arrive in sequence can be calculated by dividing qInAccDelay by qInMsgCounter. Similarly, the *rate* is

InMsgCounter divided by the time since qInActTime. In addition, one may use these objects to compute other statistics. For example, the mean bandwidth usage for messages in sequence can be calculated by dividing qInMsgVolume by the difference between qInLstMsg and qInActTime. These values are averages when the window size is equal to the duration of the connection so far. For other window sizes, the QoS statistics are stored by qProg group objects.

#	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 delays of all messages that arrived in sequence
06	qInAccJitter	Counter32	Average jitter of messages that arrived in sequence

Table 4.5: Operational Behavior Statistics Objects for Messages that Arrive in Sequence

#	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[6]	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	qProgTransPort	INTEGER	Transport layer address of the port in which the metric is being measured
07	qProgLocalAddr	OBJECT IDENTIFIER	Address of the machine where qProgTransPort is located
08	qProgRemoteTransPort	INTEGER	Transport layer addresses of the port connected to qProgTransPort
09	qProgRemotetAddr	OBJECT IDENTIFIER	Address of the machine where qProgRemoteTransPort is located

Table 4.6: Programmable Group Objects

[6] 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.

4.7 QoS MIB Data per Programmable Metric

The goal of this group is to store QoS metrics programmed by applications. Table 4.6 illustrates the objects in this group. Notice that based on the group objects 6 through 9, for example, QoS managers can trace the communication over which the QoS metric is being measured and find out more information on the corresponding qOut and qIn group objects.

4.8 Challenges in QoS MIB Instrumentation

In QoSME, QoSockets and QoSOS run time systems automatically monitor the QoS delivered and update QoS MIBs accordingly. This section overviews the implementation and real time characteristics of the QoS MIB instrumentation in QoSME.

The design of QoS MIBs instrumentation is aimed at fulfilling the following goals:

- Monitoring and collection should incur minimal runtime overhead to preserve real time properties of QoS demanding activities.
- The information collected should be available concurrently to the application being monitored and to other applications involved in managing QoS delivery.

QoSockets and QoSOS adopted a *shared memory based design* to fulfill the criteria above, as illustrated in Figure 4.4. The superimposed squares represent application threads of execution. The cubes represent portions of shared memory, one for each QoS MIB group. The rectangular name tags indicate the QoS MIB group stored in the shared memory fragment. 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. 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.

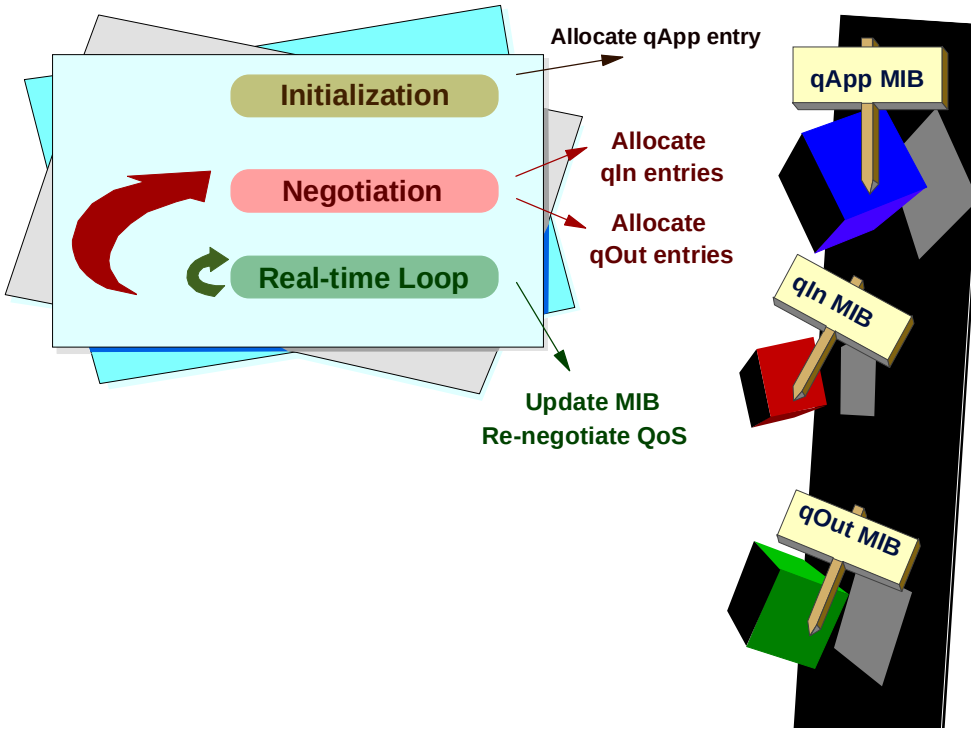

Figure 4.4: Shared Memory Design for QoS MIB Data Collection

The memory allocation is based on a general characterization of the execution flow of real time multimedia applications. The general characterization 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, such as establishment of QoS demanding connections or allocation of processing resources.
- *Real time loop:* involves the actual real time processing and transmission of data, such as sampling or 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 latter case, applications return to the negotiation phase, request new QoS, and engage the real time loop again.

It is important to emphasize that the only blocking activity performed for QoS monitoring is the allocation of entries in the shared memory space, which does not occur in the real time loop. Allocation of memory is blocking because an inter-process synchronization mechanism (that can cause unpredictable delays) must be used to coordinate book keeping of available memory modules.

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, which are only read by others. Similarly, QoS MIB SNMP agents have permission to write only to objects that cannot be written by applications. MIB objects may, however, be read or written concurrently. In summary, QoS MIB updates have bounded computational cost (the cost of a write operation in a shared memory position).

Shared memory areas for QoS MIB data are accessible by all applications running on a system. However, shared memories are not robust to system failures. It is the responsibil-

ity of the QoS MIB SNMP agent to make backups of QoS MIB data and to garbage collect entries.

The shared memory based design for QoS MIB data collection is particularly suitable for multi-processor architectures where shared memory blocks are visible to applications running on 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, one or more processors can be dedicated only for the QoS MIB SNMP agent when the number of SNMP requests is very high.

4.9 Conclusions

This chapter presented an architecture for QoS management using QoS MIBs. QoS MIB data identify how communication and processing resources are allocated and utilized by applications. Applications use QoS MIB data to detect QoS violations and adapt accordingly. SNMP agents in the architecture provide QoS MIB access to SNMP managers that may use this information to manage resources according to the QoS delivered to applications. QoS MIB objects also include control information to coordinate QoS management between applications and SNMP managers.

QoS MIBs store statistics on universal and new application programmed metrics. They store a fixed set of information for each application running, for their inport, and for their outport connections. Additionally, applications can dynamically add QoS metrics to QoS MIBs. QoS MIBs convey enough information to characterize all QoS metrics that can be defined using the formalism in Appendix A.

Chapter 5

Experiments with QoSockets: Applications and Performance

5.1 Introduction

5.1.1 The Problem

This chapter addresses two questions:

1. *What applications can gain by using QoSockets?* The goal is to identify specific features in QoSockets that ease the implementation of distributed multimedia applications.
2. *What is the performance overhead in using QoSockets?* The goal is to assess if there is significant overhead or loss in performance due to QoSockets.

This study applies also to the QuAL communication constructs since the latter are implemented using QoSockets.

5.1.2 Main Results

A team of students[1] and researchers[2] at Columbia has implemented four multimedia

[1] Jian Ping Chen, Mikhail Kishlev, Anatoly Korolev, Judy Chih-Chi Su, Robert Shteynfeld, and Aruchunan Vaseekaran.

[2] Sanjay Kumar Jha and Margarita Safonova.

applications using QoSockets. These applications use important features of QoSockets such as protocol independence and automatic QoS monitoring. One of the tools is an extension of the Mbone [Kumar 95] *net video* (*nv*) tool [Frederick 94]. This latter experiment demonstrated the ease of portability of an existing tool to QoSockets by simply replacing the sockets API in nv with the QoSockets API.

QoSockets performance was assessed in terms of overhead and throughput. The experiments have been conducted over UDP/IP, TCP/IP, and AAL. The overhead per message is constant for all transport protocols evaluated. This means that the cost of using QoSockets is diluted as the message size increases. The throughput approaches the one of the underlying protocols as the message size increases. The reason for that is that the cost of generating messages (that is, of copying them from the user to the kernel spaces) is much higher than the cost of executing the QoSockets API and protocol processing.

One should notice that the QoSockets overhead would have existed anyway in a typical multimedia application. The difference is that the functionality offered by QoSockets would have been scattered through the multimedia application which can lead to further inefficiencies, increased implementation overhead, and logic errors. QoSockets concentrate QoS negotiation and monitoring in a single location which may lower the incidence of logic errors and improve the implementation efficiency (potentially with the help of special hardware devices).

5.1.3 Chapter Organization

The reminder of this chapter is organized as follows. Section 5.2 describes applications implemented using QoSockets, Section 5.3 measures QoSockets performance, and Sec-

tion 5.4 concludes.

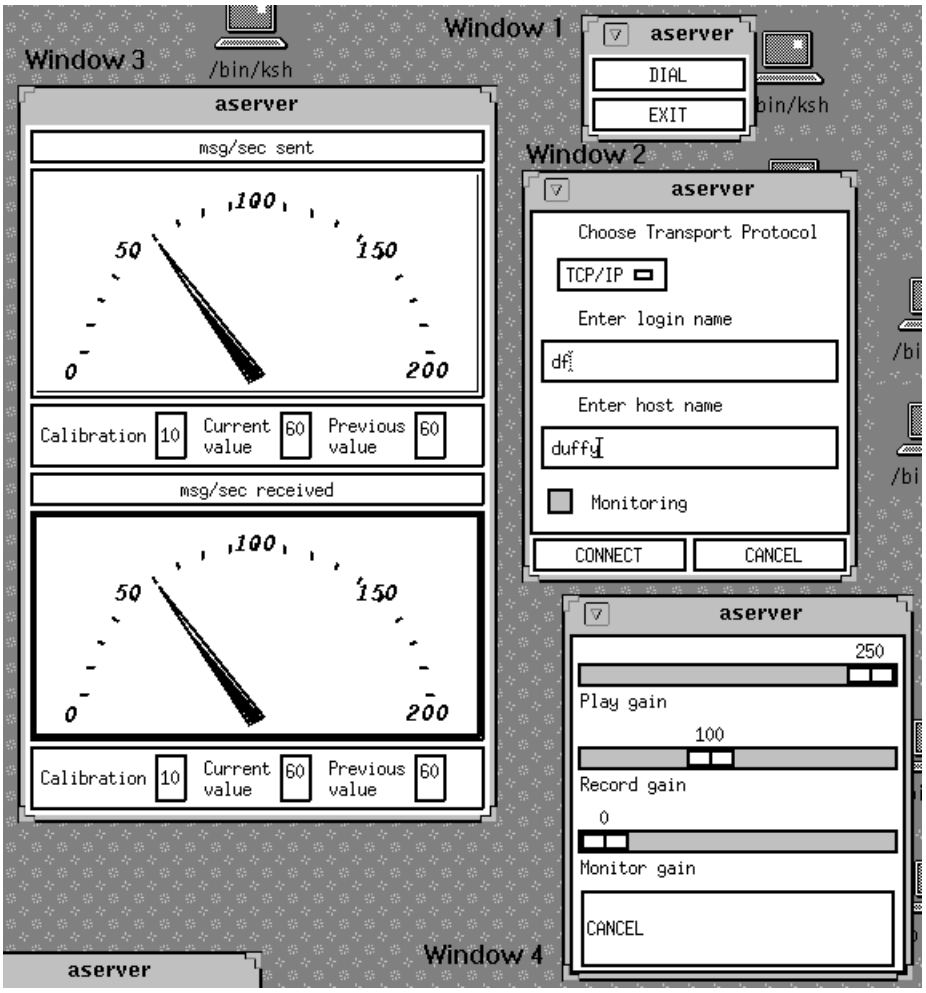

Figure 5.1: Audio Tool from the Caller Side

5.2 Applications

This section reports on four applications developed to experiment with the QoSockets library over the Internet. The applications are:

- An audio tool for telephone-like service.
- A video tool for video communications.
- An integrated video and audio conference tool.

- An extension of the Mbone nv tool for QoS monitoring.

The following sections explain each of these tools in more detail.

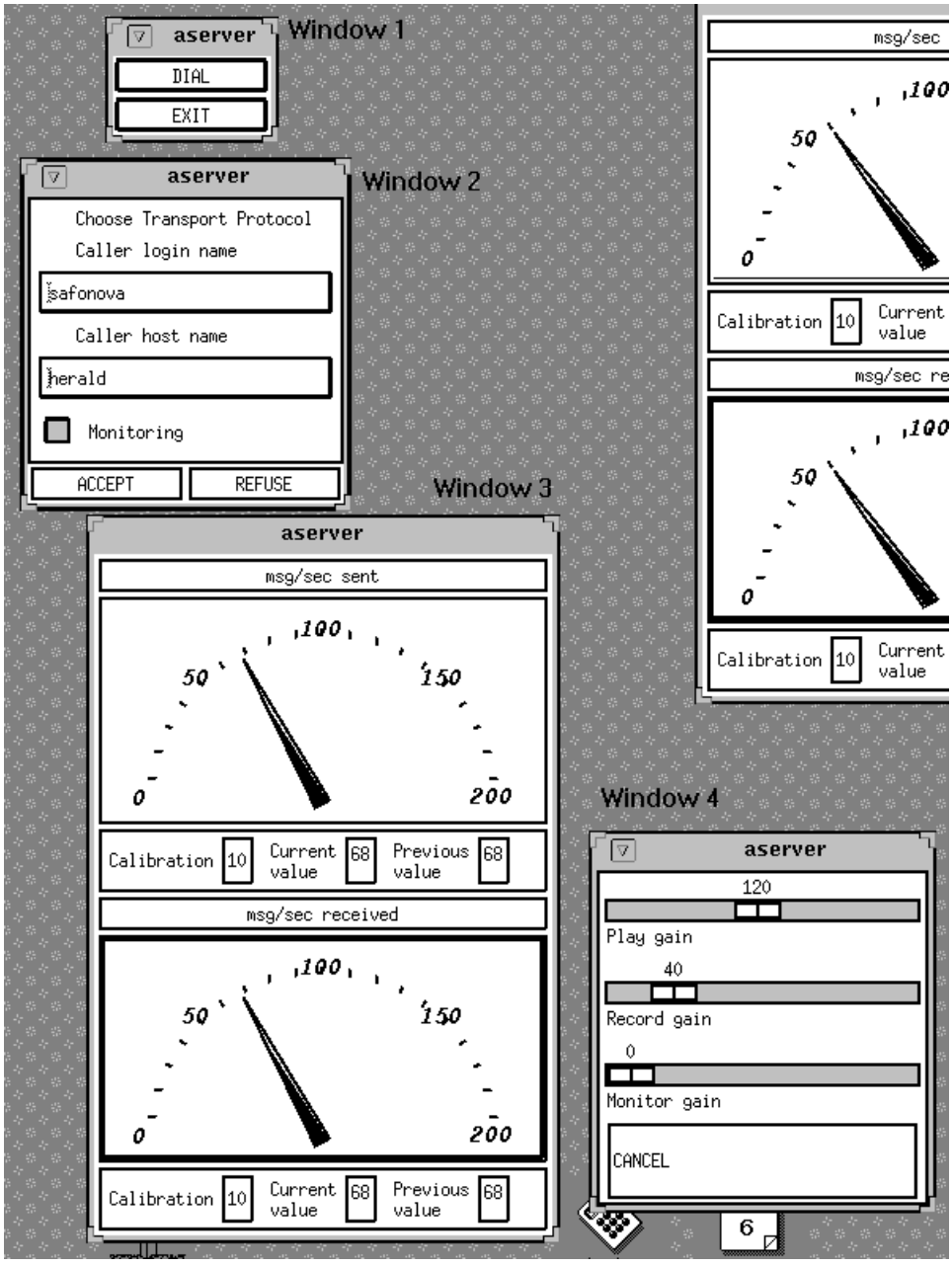

Figure 5.2: Audio Tool from the Callee Side

5.2.1 Audio Tool

The goal of the audio tool is to provide point-to-point voice communications over the Internet. Figure 5.1 (Figure 5.2) depict the caller (callee) windows after the call has been accepted. The protocol to initiate the call is the following. The caller starts with Window 1. It has two options: *DIAL* or *EXIT*.

When the user clicks *DIAL*, Window 2 starts to ask for the desired callee information, monitoring option, and protocol to use. The user enables monitoring when s/he clicks the box next to the *Monitoring* option.

The available protocol options are "*any*", "*TCP/IP*", "*UDP/IP*", "*ST-II*", and "*AAL*". When the user chooses "*any*", the QoSockets runtime will automatically choose the best fit protocol given the particular system configuration. In Figure 5.1, the caller decided to operate on TCP/IP[3] and to connect to callee "*df*" at machine "*duffy*".

Once the data in Window 2 is complete, the user may *CONNECT* or *CANCEL*. When connecting, Windows 3 and 4 start. Window 3 contains gauges that reflect the current sending and receiving rates (in messages per second). It displays in the bottom the current (*Current value*) and previous (*Previous value*) sampling window rates. The user may set the gauge resolution (distance between marks) by setting the *Calibration* field. The current message rate is 60 messages/s at the caller side and 68 messages/s at the callee side[4].

Window 4 contains some sliding controls for the local audio device volume when re-

[3] The TCP/IP option is reasonable in this case because the connection takes place in a local area network. In this particular case, it is possible to wait for re-transmission within the inter-voice sample delay (125 μs). In metropolitan or wide area networks it would be better to establish an UDP/IP connection and lose late voice samples.

[4] Sometimes the receiving rate during a particular window may be higher than the sending rate due to messages left over from the previous windows.

cording (*Recording gain*) and playing (*Playing gain*) or both (*Monitoring gain*).

The callee side contains similar windows, except for Window 2 that has mainly a display function. It displays the caller information and the protocol in use. The callee may choose to *ACCEPT* or *REFUSE* the call. It may also enable monitoring at its side.

QoSockets simplified the following features of the audio application implementation:

- *Protocol transparency*. The code to implement the multiple supported protocols uses the same QoSockets API. Each protocol chosen by the user becomes an argument of the API. This facilitates support of multiple protocols and portability to new ones.
- *Monitoring automation.* The application code specifies the necessary QoS using QoSockets API. The QoSockets runtime collects monitoring data in the QoS MIBs which feed the gauges with the current sending and receiving rates. The whole monitoring implementation becomes simple QoS MIB value retrievals.

There are many directions to extend the audio tool with little effort:

- *Monitoring of other QoS parameters*. Other QoS metrics collected in the QoS MIBs may be displayed. Examples include jitter and loss.
- *Interaction with SNMP managers*. The simple fact that this tool runs on a particular system may enhance the knowledge of SNMP managers about current network conditions from application perspective. They may adapt to QoS degradation by allocating alternative paths or destroying interfering applications.
- *Application adaptation*. Applications may also access monitoring information and adapt to QoS degradation. For example, they may re-negotiate the QoS or switch

to text interactions.

The same advantages can be observed for most of the applications in this chapter.

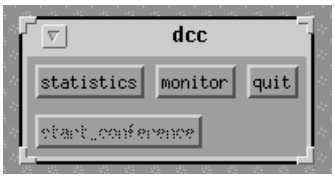

Figure 5.3: Audio Tool from the Caller Side

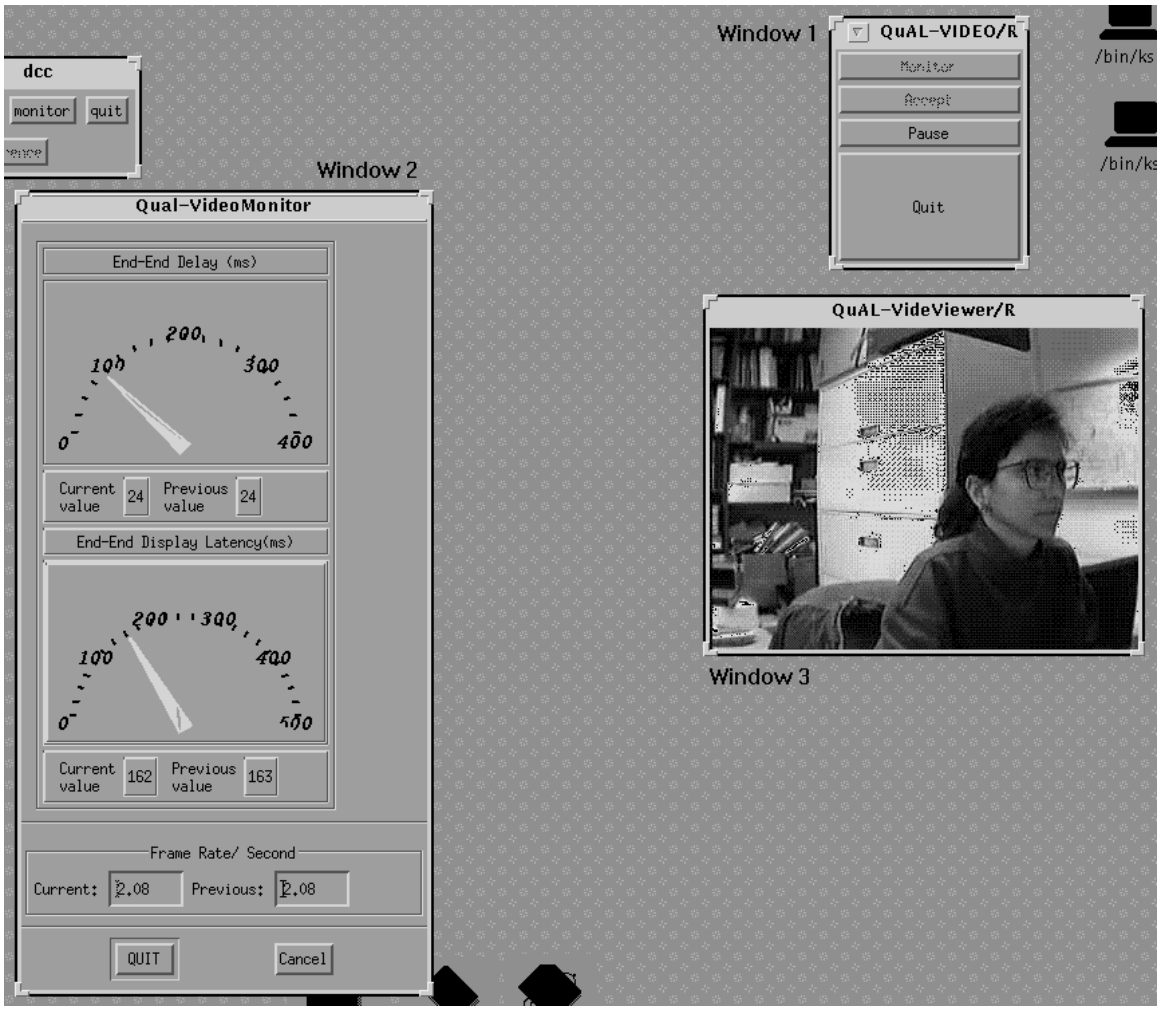

Figure 5.4: Audio Tool from the Callee Side

5.2.2 Video Tool

The video tool provides video point-to-point connection. The caller side is depicted in Figure 5.3 while the callee one in Figure 5.4. The caller window and Window 1 at the callee side are used for the connection establishment protocol. It is very similar to the one in the audio tool (Windows 1 and 2 in Figure 5.1 and Figure 5.2) and is omitted in what follows.

Once the callee accepts the call, monitoring starts at its side. The monitoring gauges in Window 2 measure *End-to-End Delay* and *Display Latency*. The latter refers to the period of time from message transmission until it is displayed at the destination. Using the notation in Appendix A, it is:

$$display^*: T \times T \times \Phi \rightarrow R$$

$$display^*(b_w, e_w, \Pi_S) = \frac{\sum_{\forall k \text{ in } \Pi_S[b_w \le p \le e_w]} (p_k - s_k)}{\left|\Pi_S[b_w \le p \le e_w]\right|}$$

The transmission and reception rates in messages per second are displayed in the boxes underneath the gauges.

Finally, Window 3 displays the video frames.

This application is similar to the audio tool and thus profits from the same QoSockets features. One may observe, nevertheless, that both applications may decide to adapt to QoS degradation using very different strategies. For example, the audio tool cannot operate if the bandwidth dedicated to the connection is lower than 64 Kbits/s or if there is significant jitter between frames. It may need to switch to another media such as text when it detects such situation. Nevertheless, the video tool may recur to other solutions such as

lowering the frame rate from 30 frames/s to 15 frames/s or switching to black-and-white frames. The final quality of the video interaction decreases, but it is acceptable in many situations. These two tools illustrate how the application semantics and the recovery strategy are intertwined. QoSockets help to provide the necessary control on the recovery strategy to the application developer.

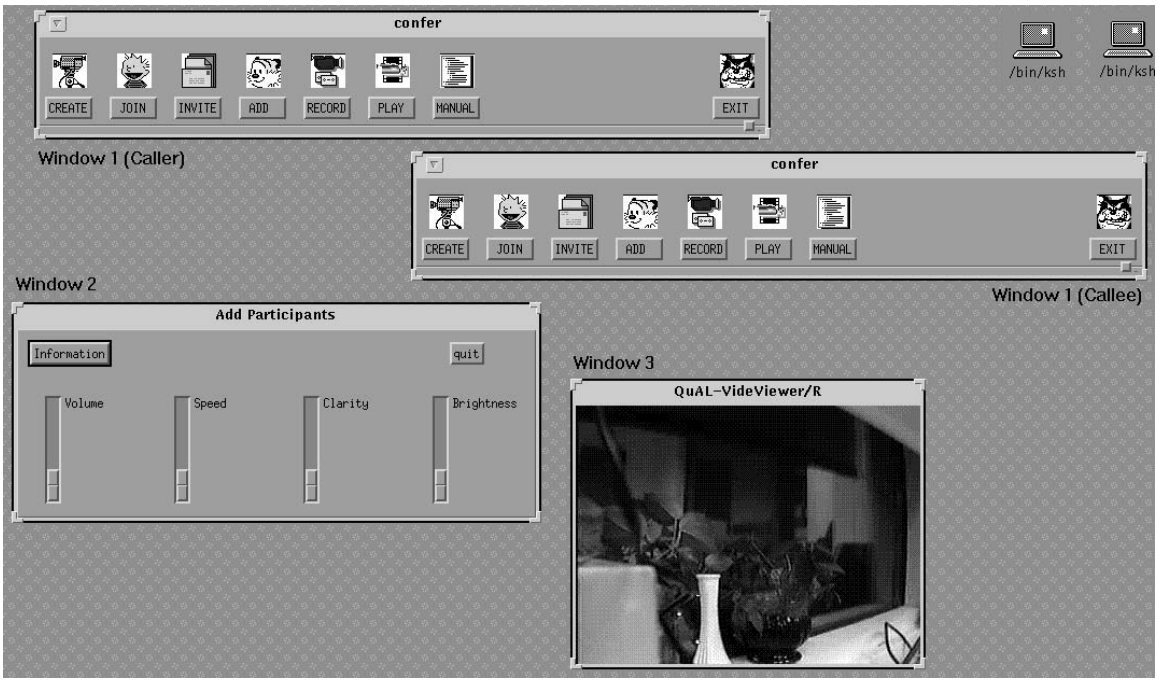

Figure 5.5: Video Conference Tool (Callee or Caller)

5.2.3 Integrated Audio and Video Conference

The video conference application is a tool for audio and video exchange among participants in a session. Figure 5.5 displays one of the sides in the communication (the other one is similar). Window 1 provides controls to *CREATE* a conference or *JOIN* an existing one. Additionally, one may *INVITE* other participants and *ADD* them when they accept the invitation. The conference may be *RECORD*ed and *PLAY*ed back at a later time. Fi-

nally, there are controls to *EXIT* the conference or to read the *MANUAL* of the application.

A participant will also have the controls in Window 2 available. Most control the screen (*Clarity* and *Brightness*) or audio devices (*Volume*). But one of them can control the sampling and transmission rates (*Speed*). The idea is that when the quality of the transmission deteriorates in Window 3 (where the video is displayed), one may adapt by sliding the *Speed* control. For example, when one notices too much loss, one may reduce the message transmission rate.

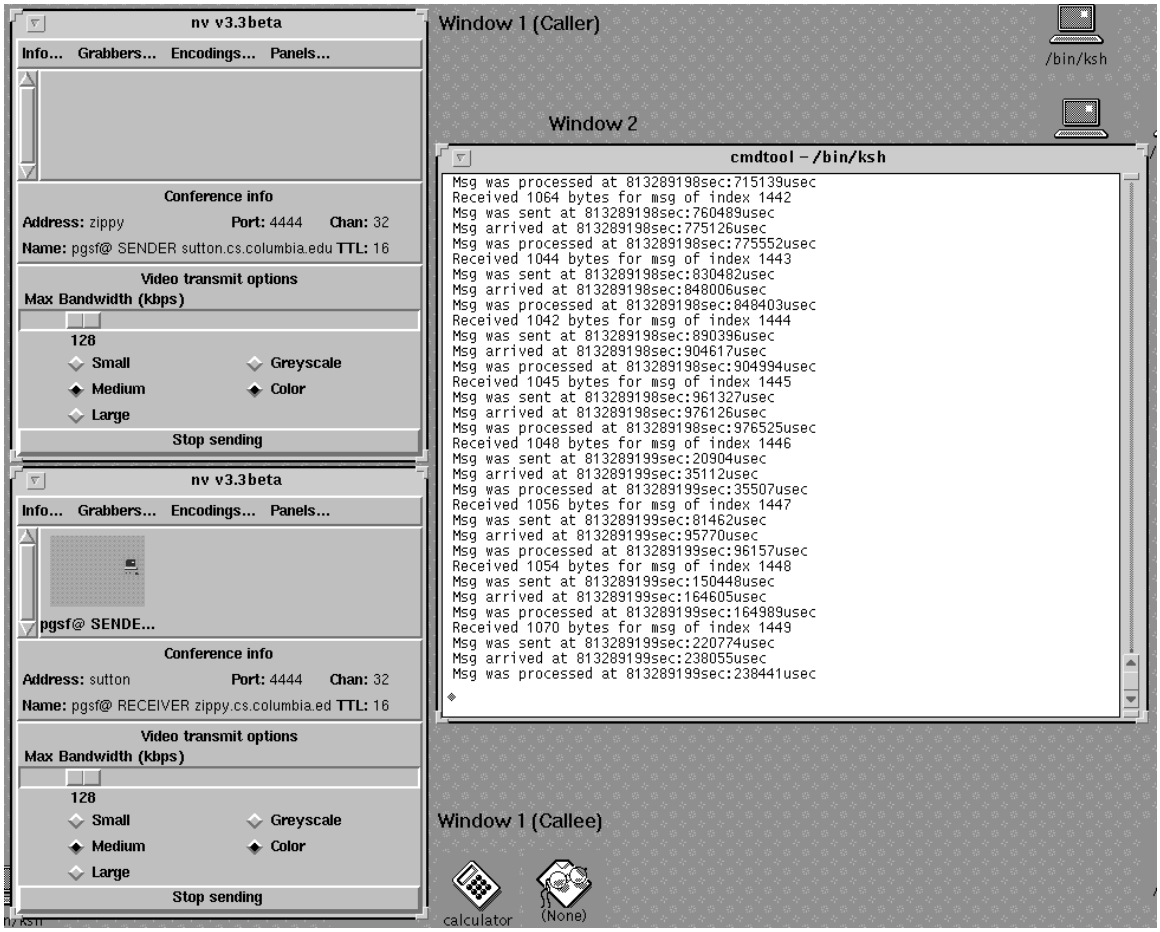

Figure 5.6: The nv Tool

In addition to the features illustrated for the audio and video tools, this one illustrates how an application developer may use QoSockets to change the QoS allocation to adapt to network conditions. In this particular case, the application developer gave the user direct control over the transmission rate adaptation to network conditions.

5.2.4 QoS Monitoring Extension to the Mbone Net Video

The goal of this experiment was to extend an existing distributed multimedia tool with the functionality that QoSockets offer. The one chosen was the nv tool that implements video conference on the Internet Mbone. The original nv was implemented using the sockets API.

The extension work involved replacing the sockets API with the QoSockets APIs. As a result, the extended nv can be ported among multiple platforms and can report on the QoS in each of its connections.

The extended tool is depicted in Figure 5.6. Window 1 is the traditional nv screen (the details can be found in [Frederick 94]) to open and view video sessions. Window 2 is a QoS trace of the application. It reports on the many QoS parameters such as loss, throughput, etc.

The next step is to implement an extended version of this tool where the user may specify the QoS metric they want to monitor using a special equation editor. The specification is translated into the QoSockets API. Finally, a performance window displays the specified metric. The latter may be shaped as a graph or as a gauge similar to the one in Figure 5.1.

5.3 Performance

QoSockets define a new layer of functionality on top of the sending and receiving protocol API and thus increases its overhead. Such layer includes the following functionality:

- *Time stamping*. This step retrieves the time from the local clock and stores it in the message. The time stamp has 8 bytes.
- *Index generation*. A unique index 4 bytes long based on the sending time is generated and stored in each message of a stream.
- *Conversions from/to External Data Representation* (*XDR*) [Sun Microsystems 87]. The message header (index and time stamp) are encoded in the XDR format for compatibility across different architectures.
- *QoS MIB updates*. Message transmissions spawn multiple QoS MIB updates. The computational cost of such updates is bounded. The runtime updates the qOut (qIn) table upon transmission (reception). It updates the qProg table only when the windows end, but these events happen off-line with respect to transmissions or receptions.

The study of QoSockets performance has the following goals:

- *Study QoSockets overhead.* The goal was to understand the overhead per message transmission.
- *Study QoSockets throughput.* The goal was to understand how many messages per second could be sent using the QoSockets API.

One approach to find the exact throughput and overhead would have been to examine

the exact CPU time each instruction would incur in a typical machine. Such cost is constant 200 μsec on a SPARC 20 workstations using Solaris 2.4. But, this measure depends on the particular architecture and implementation technique. Additionally, it does not give insight about operations in real situations where OS schedulers, background traffic, and network loads may affect performance in a manner that is difficult to predict.

The approach chosen was to measure the sending[5] time through the QoSockets APIs and through the other stacks APIs. The UDP/IP and TCP/IP measurements were conducted between SPARC 20 machines running Solaris 2.4, while the AAL measurements were conducted between SPARC 10 machines.

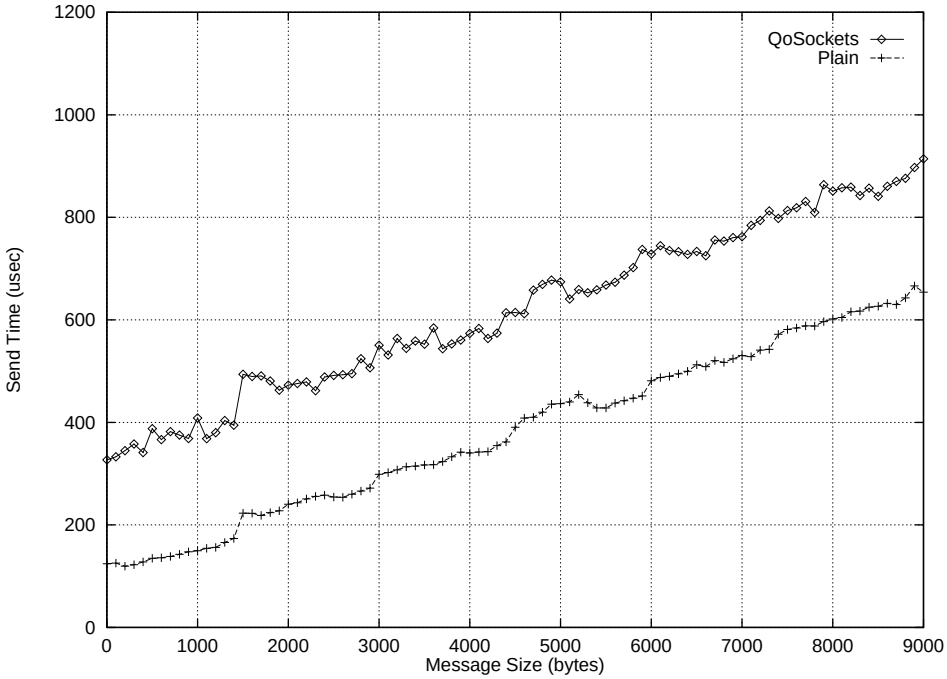

Figure 5.7: Comparison of Sending Times over QoSockets and UDP/IP

The size of the message payloads vary from 0 to 9000 bytes. This range was chosen

[5] The performance (overhead and throughput) of the receiving side is similar since the QoSockets API at both sides have very similar execution flows.

because the maximum frame size that the FORE ATM SBA-200 network adapter (with driver version 2.2.6) can handle is 9188 bytes and one of the goals was to compare the overhead for all transport protocols tested using the same message sizes.

5.3.1 Overhead

The overhead computed per message is depicted in Figure 5.7 through Figure 5.9 for many transport layers protocols and for QoSockets. The figures depict the sending time for a range of message sizes. These results were derived by sending one message, computing its delay, and then waiting to send the next message. This technique avoids queueing overheads for any protocol or card.

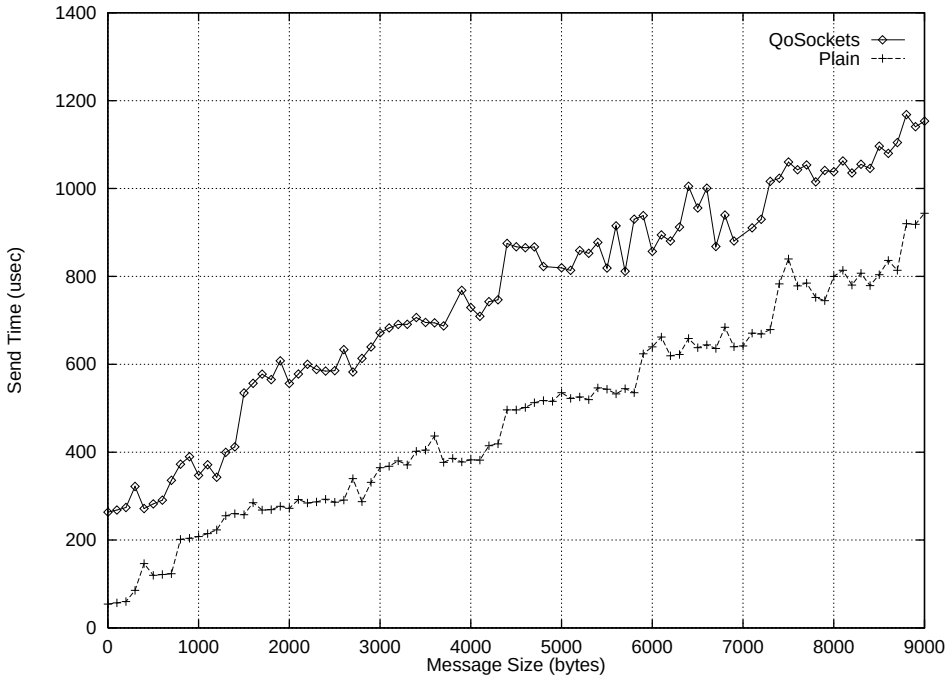

Figure 5.8: Comparison of Sending Times over QoSockets and TCP/IP

One can see that the overhead is almost constant throughout the message size spectrum. It dilutes as the message size increases, but is nonetheless significant. The reason is

that most transport protocols studied use very efficient code implementation (many times in Assembly language).

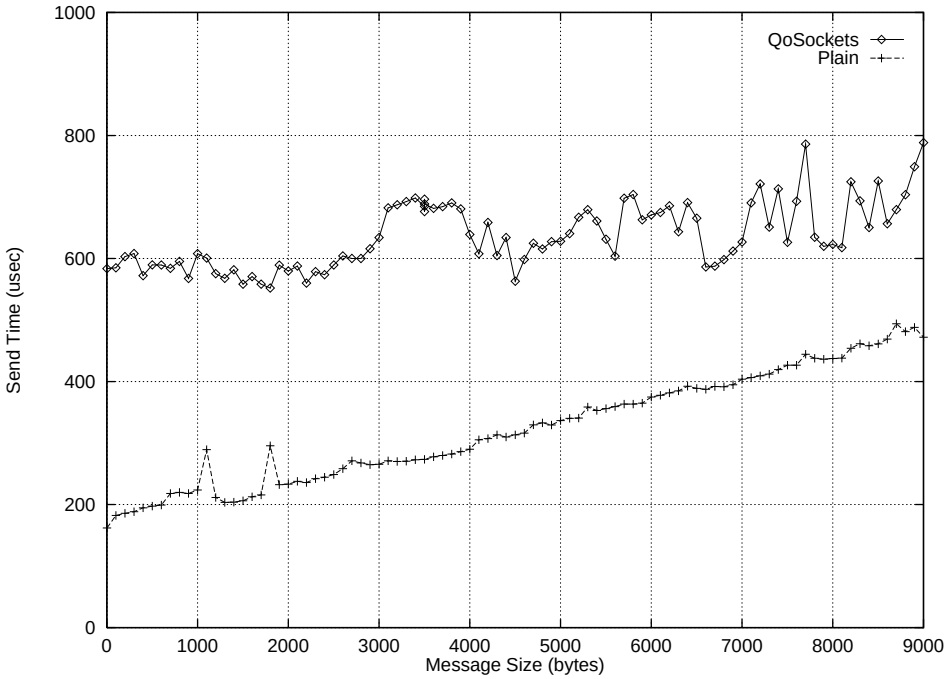

Figure 5.9: Comparison of Sending Times over QoSockets and ALL

QoSockets were implemented in C as a first prototype and thus it suffers the inefficiencies of the compilation process. The potential for future performance improvement is big because the instruction in QoSockets (like time stamping, index generation, functionality for MIB updates) are amenable for implementation using dedicated or customized hardware. Finally, a significant amount of pipeline is possible between QoSockets and message transmissions by processing timestamping, index generation, and MIB updates of previous of new messages while older ones are being transmitted.

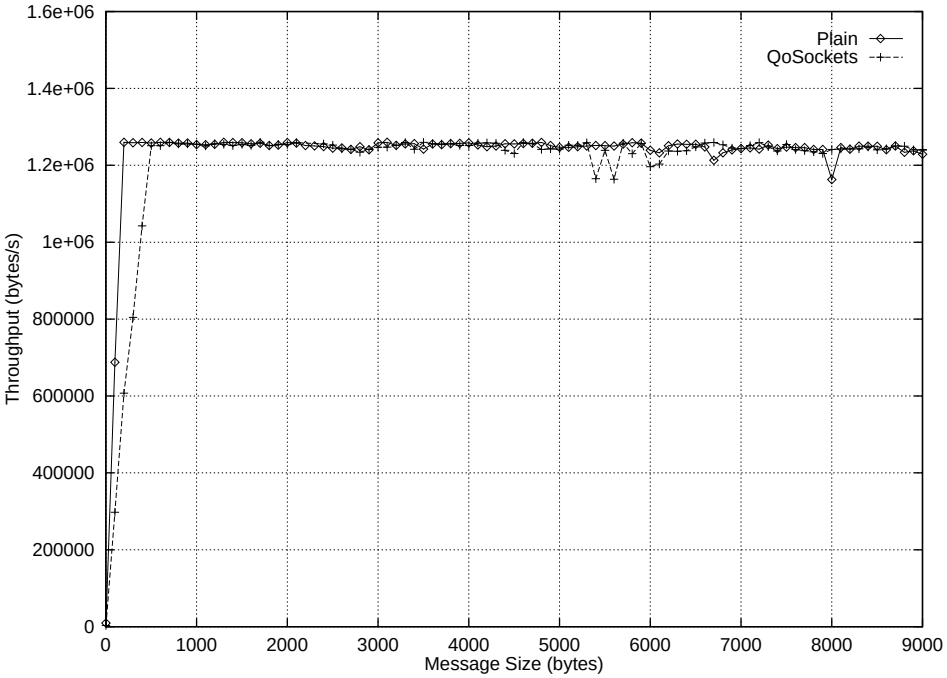

Figure 5.10: Comparison of Throughput over QoSockets and UDP/IP

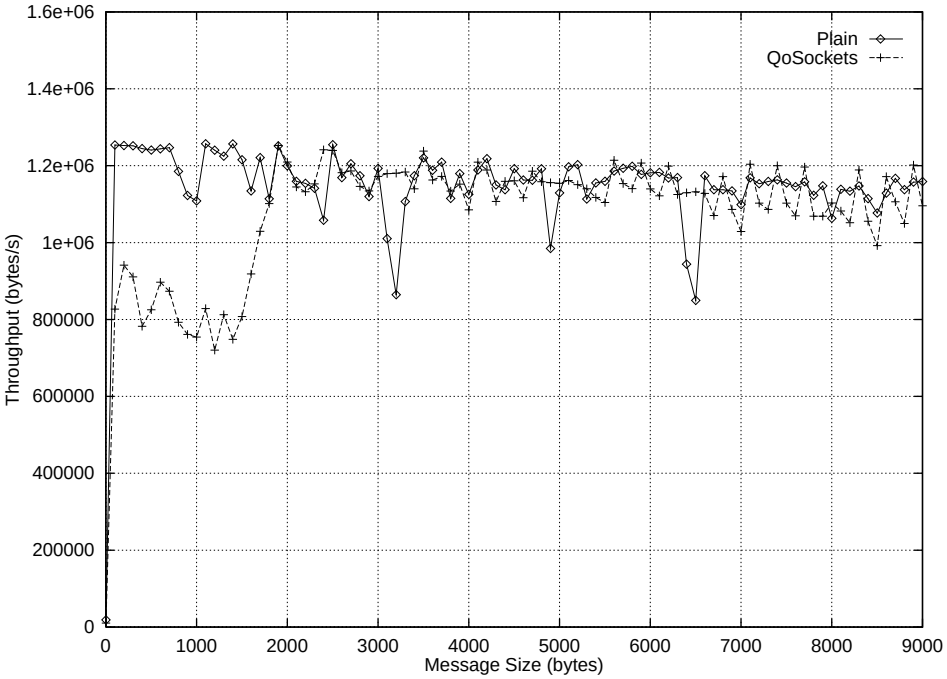

Figure 5.11: Comparison of Throughput over QoSockets and TCP/IP

To conclude, it is important to understand that the functionality in QoSockets would

have to be implemented in distributed multimedia applications anyway and, as a consequence, the overhead would exist in any case. The advantage of QoSockets is that future implementations can fine tune the code to make it efficient and increase the overall performance of QoSockets applications.

5.3.2 Throughput

The throughput is depicted in Figure 5.10 through Figure 5.12 for multiple message sizes. These measurements were conducted by sending messages as fast as possible with eventual queueing for protocols and outgoing cards.

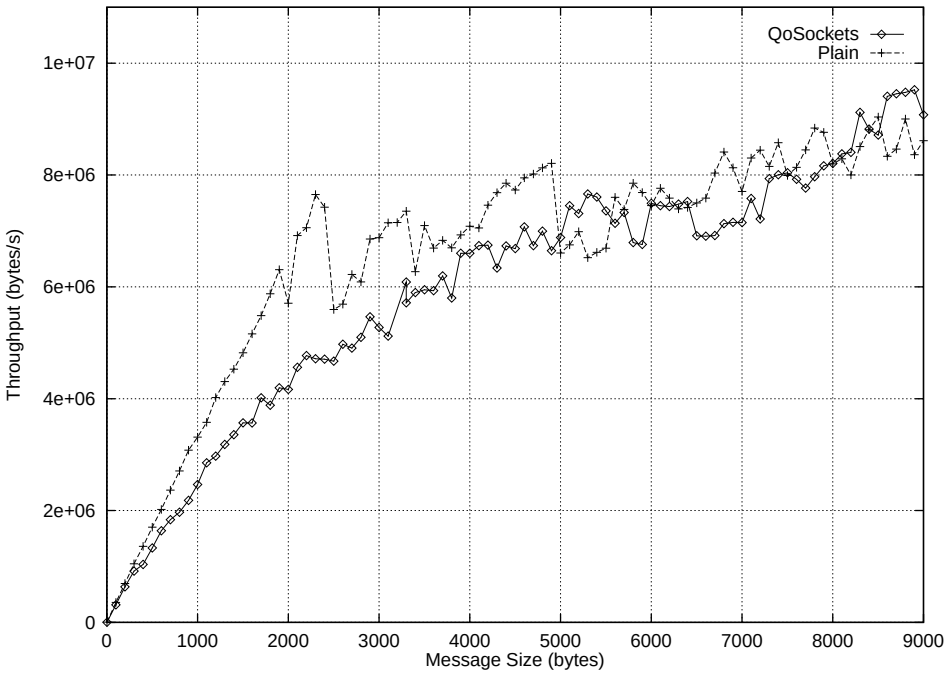

Figure 5.12: Comparison of Throughput over QoSockets and AAL

One can see that, as the message sizes increase, the throughput quickly becomes comparable to the one of the underlying transport layer. The reason is that the time to copy messages between user and kernel spaces is much larger than the time to time stamp,

process, and then send them through the protocol stack. Thus, the final throughput is dictated by the message copy process rather than QoSockets processing.

5.4 Conclusions

Two sorts of experiments were conducted using QoSockets. The first was the implementation of multimedia applications using the library to evaluate its advantages. As expected, they simplified many aspects of the implementation of such applications, including multiple protocol support, automatic QoS monitoring, and portability.

The second was the evaluation of the overhead and throughput of the QoSockets implementation. It was shown that the overhead is not negligible (200 μsec in a SPARC 20) but it is constant for any message size. The throughput is not considerably affected because the typical time to generate a message is much larger than the time to process QoSockets API and protocols.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Application level QoS management plays a central role in distributed multimedia applications. Current internetworks may offer strict QoS guarantees within particular domains, but end-to-end guarantees may involve complicated re-engineering of the current state of the art. On one hand, the semantics of multimedia applications running on such internetworks require checking of the QoS effectively delivered by the network in order to adapt to QoS failures. On the other hand, application QoS management can involve tedious, repetitive, and error-prone tasks. In addition, it is useful to integrate application management with traditional network management standards such as SNMP. This integration would facilitate coordination between SNMP managers and applications to resolve QoS degradation issues.

This thesis proposed the *QoS Management Environment* (*QoSME*) that automates application QoS management. It consists of the QuAL programming language, the QoSockets and QoSOS APIs, and QoS MIB definitions with associated SNMP agents. The main contributions of this work are:

1. *Language level approach to QoS specification.* The QoS required by applications

can be specified using QuAL language constructs that are later compiled into the underlying transport protocol and OS services. In addition, QoS management constructs can specify how to monitor, analyze, and control the requested QoS. There are a few advantages to this approach: (1) it provides a unified API across heterogeneous transport protocols and OSs; (2) it permits automatic compilation of QoS specifications into transport protocol parameters; (3) it promotes application-specific handling of QoS violations; and (4) it permits checking of inconsistencies between QoS requests of communicating applications.

2. *Automatic QoS monitoring.* When QuAL applications are compiled, QoSME automatically generates instrumentation to save application QoS performance statistics in QoS MIBs. Additionally, QoSME monitors the QoS effectively delivered and invokes customized exception handlers upon violations. The main contributions are: (1) automatic generation of monitoring instrumentation; and (2) collection of QoS performance profiles into QoS MIBs.
3. *Integration with standard network management.* SNMP agents within QoSME disclose the contents of QoS MIBs to external SNMP managers that may use them in performing management tasks. This approach introduces the following contributions: (1) SNMP managers can analyze application QoS performance; (2) applications and SNMP managers may coordinate their efforts in solving QoS violations; and (3) management can focus on application level properties.

6.2 Future Work

6.2.1 High-level QoS Libraries

How to build high-level libraries for QoS management? The goal is to associate new high-level operators on streams of messages. For example, operator *sync(s1,s2)* synchronizes streams *s1* and *s2*. Operator *half(s)* reduces by half the number of messages in stream *s* by dropping every other message. Given these operators, one can build new streams from primitive ones that can entail completely new QoS characteristics. A preliminary classification of such operators follows:

- *Filters*. These operators eliminate messages from a stream based on certain criterion. For example, *drop(s,* t_a-t_s*<10 ns)* eliminates all messages in stream *s* with end-to-end delay of 10 ns or more.
- *Combiners.* These operators combine streams based on some criterion. For example, *interleave(s1, s2)* generates a new stream that contains message with index *i* of *s1* immediately followed by message with index *i* of *s2*, for all indexes *i*.
- *Permutators*. These operators change the order of the messages in a stream according to some criterion. For example, *sort(s,*t_a*)* sorts all messages in *s* according to their arrival times.

These operators can provide a high-level interface for the specification of how to generate new streams from old ones.

QuAL, QoSockets, and QoSOS provide an infrastructure on which such libraries may be built. One can program them using customized QoS metrics and exception handlers.

For example *drop(s, t_a-t_s<10 ns)* is implemented by specifying that the QoS metric end-to-end delay should be smaller than 10 ns and that otherwise the exception handler should drop them.

The challenges in this research are:

- *How to define a more complete classification of the operators?* The classification proposed in this section is preliminary. More study is necessary to understand what sort of operators the target applications would need.
- *What are the primitive operators in each class?* One interesting approach is to define a set of primitive operators for each class and constructors to build more elaborate ones.
- *How to build a library of useful operators in each class?* The goal is to implement the operators efficiently using the QuAL, QoSockets, and QoSOS constructs.

6.2.2 Pricing

How should network services be requested and charged? There are three preliminary approaches that one can propose:

- *Flat rates.* In this case, the service provider offers a limited set of options that the subscriber can use. For example, users interested only in sending e-mail may select maximum allowed bandwidth per month without QoS guarantees while users that need multimedia video and voice may select services with strict QoS guarantees on pre-set bandwidth per month. The advantage of this system is that each user knows what the bill will be at the end of the month. The disadvantage is the lack of flexi-

bility if services not anticipated become necessary.

- *Charge on-demand.* In this case, the service provider will charge for the services used. For example, if a user decides to participate in a multimedia exchange, s/he will be charged accordingly by the end of the month. The advantage of this system is that all services are available as long as the user is willing to pay for them. The disadvantage is that the final bill may be hard to predict.
- *Special rates.* This is a blending of the previous two approaches in which the user specifies what its usage pattern normally is and the service provider may offer special deals as long as the user stays within the initial arrangement. The user may still use other services or violate the initial arrangements, but the prices for services not pre-arranged are considerably higher.

For all these approaches there must be a mechanism that will collect QoS specifications by individual users. Such information will allow the billing infrastructure to: (1) validate service usage according to pre-agreed arrangements, and (2) compute actual usage to decide final charges. The QoS specification mechanism can be implemented in QoSME.

In addition, a billing infrastructure can use QoSME to collect usage statistics per application. These statistics can be stored in QoS MIBs. Collection systems can use the MIBs to decide how to charge each application according to its usage.

The challenges in this research are:

- *What is the best MIB design to include all the necessary billing information?* The QoS MIBs may not contain all the information necessary for billing purposes. The goal is to find the necessary supplementary fields that would yield such informa-

tion. QuAL contains operators that enable dynamic addition of new fields to QoS MIBs.

- *How can the user specify its QoS needs?* QoSME QoS specification constructs may be too detailed for usage specification. The goal is to create a higher-layer specification mechanism that is then compiled into QuAL constructs.
- *How to develop charging mechanisms on QoSME?* Charging seems to be a direct extension of the monitoring services QoSME intends to provide. One may develop a charging system as an extension to QoSME.

6.2.3 Integrated Network and Application Management

How to develop tools and strategies that can profit from an integrated network and application management framework? This dissertation has highlighted the importance of integrated management. It would be useful to develop tools and strategies that could best use such framework.

The challenges in this research are:

- *What sort of tools would be useful for integrated network management?* These tools include monitoring, analysis, and control activities. For example, how should the correlation between effective QoS delivered to applications and the current network status be conveyed to a system administrator?
- *What strategies can be automated?* When violations occur, many different strategies to overcome them could be automatically encoded in control tools. For example, when QoS degradation on the bandwidth penalizes one specific application,

strategies can be included to allocate alternative connections for the application.

6.2.4 Formal QuAL Semantics

How to specify the semantics of QuAL? QuAL specification may need to extend the process model as defined by Hoare [Hoare 78] because of the time-dependent characteristics of QoS specifications.

The main challenges are:

- *How to extend the process model to characterize time-related issues?* The problem is to add the minimal constructs to the original process model by Hoare in order to capture the semantics of time-dependent behaviors in QuAL.
- *How to specify QuAL using the new model?* The problem is to define every QuAL construct and mechanism using the extended process model.

Appendix A

A Model for QoS Specification

The goal of this appendix is to formally define a model for the communication QoS metrics in QoSME. It defines each universal QoS metric and shows how to define application specific ones using the model. Finally, it formalizes the concepts of violation and filter in QoSME.

A.1 Definitions

A QoS metric, in general, measures the performance of a stream of messages. A *stream* is a sequence $S = \{m_k\}_{k \in I}$, where *I* is the set of message indexes. With each m_k, one can associate a *performance signature* $\pi_k = \langle s_k, a_k, p_k \rangle$ ($s_k < a_k < p_k$), where the three components are the message sending, arriving, and processing times respectively. The arriving and processing times of lost messages are ∞. The sequence $\Pi_S = \{\pi_k\}_{m_k \in S}$ is the *performance profile* of *S*. The expression k in Π_S is true if and only if $\pi_k \in \Pi_S$. The expressions *first*(Π_S) and *last*(Π_S) denote, respectively, the index of the first and the last messages in *S*. The message $m_{k+_a 1}$ of stream S is the message consecutive to m_k according to arrival time.

A *session* is the union of non overlapping streams. That it, a session $K = \bigcup_{i=1}^{n} S_i$ must satisfy $\left[s_{first(\Pi_{S_i})}, s_{last(\Pi_{S_i})}\right] \cap \left[s_{first(\Pi_{S_j})}, s_{last(\Pi_{S_j})}\right] = \varnothing$ for all $1 \le i, j \le n$, $i \ne j$. The performance profile of a session is the union of the performance profiles of the component streams.

Most QoS measures in QuAL are defined in terms of performance signatures over windows of time. The expression $\Pi_S[C]$ denotes the set of performance signatures of Π_S that satisfy condition *C*. For example, $\Pi_S[b \le s \le e]$ denotes the performance signatures of the messages m_k of stream *S* with $b \le s_k \le e$.

Figure A.1 depicts a typical stream *S* between two communication end points. The index axis depicts the index of messages. The time axis depicts the time domain. Three time curves are shown. The first one, *s*, plots the sending time of messages and it is inherently monotonically increasing. The second one, *a*, plots the arriving time which is not necessarily monotonically increasing. That is, it is possible that $a_{k'} > a_k$ even if $s_{k'} < s_k$. This is the case, for instance, if $m_{k'}$ follows a longer path than m_k or if it is lost. This feature is depicted in Figure A.1 when *a* decreases in value. The curve *a* is strictly larger than *s*, that is, for any $m \in S$, $a_m > s_m$. The third curve, *p*, plots the processing time of messages, which is strictly larger than *a*, but not monotonically increasing either. This is the case, for instance, if messages are not processed according to a *First In First Out* (*FIFO*) schedule.

A *QoS metric* consists of a measure computed on the performance profile of a stream. That is, it is a function $qos: \Phi \to R$ where Φ is the domain of performance profiles, and *R*

is the domain of real numbers. The value of $qos(\Pi)$ is the result of computing the QoS metric qos on Π.

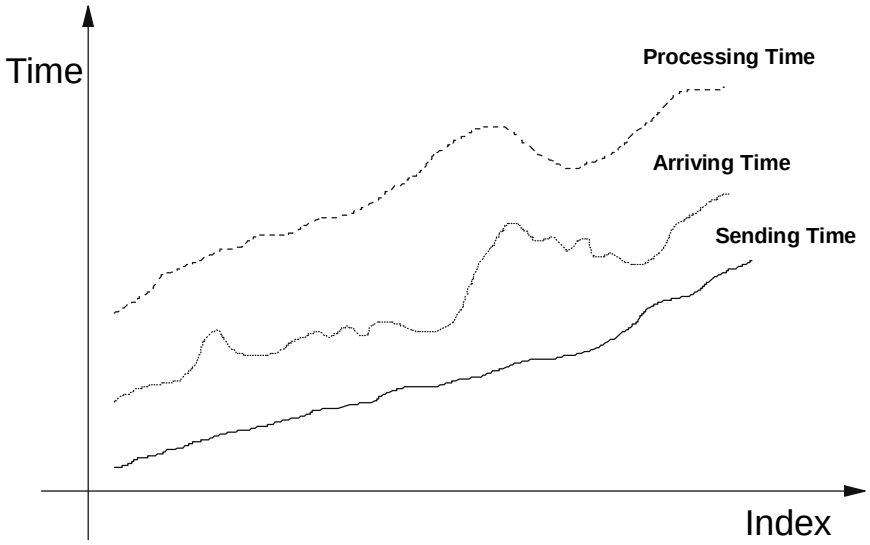

Figure A.1: Sample communication stream

One interesting issue is how to deal with QoS on multicast streams, that is, streams from one source to multiple destinations. In the model defined here, such streams are viewed as the union of multiple point-to-point (unicast) streams. The QoS of the multicast stream is defined in terms of the QoS at each component unicast stream. For example, consider a scenario where the multicast connection is distributing live video to multiple recipients with different reception equipments. It may be appropriate to request large bandwidth for users that enjoy multimedia color workstations with video capabilities while relatively small pipes to users with older gray scale workstations. The signal distributed in each stream is tailored according to the QoS supported by the end equipment (either gray scale video at small rates or full-color video at higher rates).

A.2 Some QoS Metrics Are Universal

Certain QoS metrics are universal in that they are of interest to almost any QoS-demanding application. These metrics include rate, loss, end-to-end delay, jitter, permutation, and connection recovery time. This section presents a formal definition for them.

The first universal QoS metric is *rate* that measures the mean number of messages per second received during the duration of a stream. For this purpose, it is useful to define the function $|\ |: \Pi \to N$ that counts the number of signatures (messages) in a profile (stream). For example, $|\Pi_S[t_1 \le a \le t_2]|$ is the number of messages in *S* that arrived within the time interval $[t_1, t_2]$. The formal definition of *rate* is:

$$rate: \Phi \to R$$

$$rate(\Pi_S) = \frac{|\Pi_S[a \neq \infty]|}{a_{last(\Pi_S[a \neq \infty])} - a_{first(\Pi_S[a \neq \infty])}}.$$

There is an interesting practical difficulty in computing this measure. That is, it can only be computed when the stream ends. Violations during the communication cannot be detected and corrected. There is no proper solution to this problem other than defining a new metric that approximates the behavior of *rate* for most practical purposes, even though it is not exactly the same. For now on, all approximations are distinguished from their exact counterparts by using the "*" symbol in their names.

The proposed solution is to use a *window* of time during which the rates are computed. That is, once the window starts, all statistics are reset. When the window finishes, the metrics are computed and analyzed. Immediately following the computations, the statistics are reset again and a new window starts. The process is repeated for the duration of

the stream. The same mechanism will be used for most QoS metrics. Given an interval of time $[b_w, e_w]$ and a message time-stamp β (that is, $\beta = a$, $\beta = s$, or $\beta = p$), the *qos* functions are approximated by $qos^*: T \times T \times \Pi \to R$, where T is the set of all possible time values and $qos^*(b_w, e_w, \pi)$ is the value of QoS metric qos^* calculated on $\Pi[b_w \le \beta \le e_w]$. The function *rate** is:

$$rate^*: T \times T \times \Phi \to R$$

$$rate^*(b_w, e_w, \Pi_S) = \frac{|\Pi_S[b_w \le a \le e_w]|}{e_w - b_w}.$$

The same approach is used for most QoS metrics in what follows. The exact QoS definitions are omitted because they can be inferred directly from the approximations.

The *transmission_rate* QoS metric measures the rate in which messages are sent or inserted in a stream. Its computation is very similar to the QoS metric *rate* and defined by:

$$transmission\_rate^*: T \times T \times \Phi \to R$$

$$transmission\_rate^*(b_w, e_w, \Pi_S) = \frac{|\Pi_S[b_w \le s \le e_w]|}{e_w - b_w}.$$

The next QoS metric is *loss*. A message is considered to be lost if and only if it is received more than a given timeout *t* after its sending time. It is defined by:

$$loss_t^*: T \times T \times \Phi \to R$$

$$loss_t^*(b_w, e_w, \Pi_S) = \frac{|\Pi_S[((a - s) > t) \wedge (b_w \le s \le e_w)]|}{|\Pi_S[b_w \le s \le e_w]|}.$$

The *delay* measures the average end-to-end transmission time of messages on a stream. Formally:

$$delay^*: T \times T \times \Phi \rightarrow R$$

$$delay^*(b_w, e_w, \Pi_S) = \frac{\sum_{\forall k \text{ in } \Pi_S[b_w \le a \le e_w]} (a_k - s_k)}{\left|\Pi_S[b_w \le a \le e_w]\right|}$$

The sum of the *delay* function of all messages in a stream is plotted in Figure A.2.

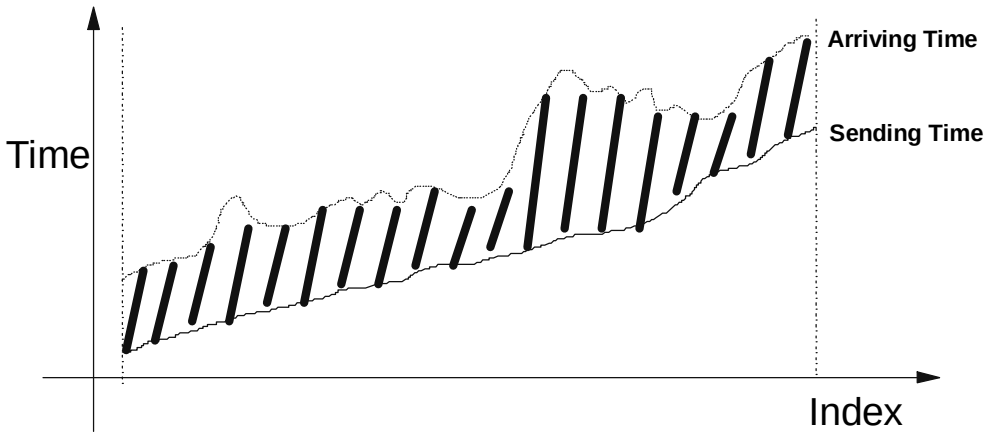

Figure A.2: Delay QoS measure

The *jitter* measures how far apart consecutive messages arrive:

$$jitter^*: T \times T \times \Phi \rightarrow R$$

$$jitter^*(b_w, e_w, \Pi_S) = \frac{\sqrt{\sum_{\forall k \text{ in } \Pi_S[b_w \le a \le e_w]} [(a_k - s_k) - delay^*(b_w, e_w, \Pi_S)]^2}}{\left|\Pi_S[b_w \le a \le e_w]\right|}.$$

The *permutation* measures how many messages arrive out of order in a stream. Formally:

$$permutation^*: T \times T \times \Phi \rightarrow R$$

$permutation^*(b_w, e_w, \Pi_S) = \left|\Pi_S[C]\right|$, where k in Π_S satisfies C if and only if:

k in $\Pi_S[b_w \le a \le e_w]$,

$\exists k'$ in $\Pi_S[b_w \le a \le e_w]$, and $k \ne k'$ satisfy both:

$s_{k'} > s_k$ (k' was sent after k) and

$a_{k'} < a_k$ (k' arrived before k).

The *recovery* measures if the time between two streams of a session is above a pre-set value *t*. The main idea is to capture the fact that several streams might need to be established during the lifetime of a session. This is the case, for instance, if a switch serving a session fails and an alternative stream is established that bypasses the damaged switch. The metric *recovery* is the period of time between successive streams of the same session. It may be formally defined using the model in similarity to the previous metrics:

$recovery^*: T \times T \times \Phi \to R$

$$recovery^*(b_w, e_w, \Pi_K) = \frac{e_w - b_w - \sum_{S \in \Omega} \left(\min\{s_{last(\Pi_S)}, e_w\} - \max\{s_{first(\Pi_S)}, b_w\} \right)}{|\Omega|},$$

where Ω is the set of streams $S \subseteq K$ such that $[s_{first(S)}, s_{last(S)}] \cap [b_w, e_w] \ne \varnothing$.

One interesting observation in the definition of *qos** is that the window size may approximate the mean or instantaneous QoS metric, depending on its size. For example, $rate^*(b_w, e_w, \Pi_S)$ approximates the mean rate of the stream when $[b_w, e_w] \to [a_{fist(S)}, a_{last(S)}]$, whereas it approximates the instantaneous rate as $b_w \to e_w$.

A.3 Some QoS Metrics Are Application Specific

Application developers can use the framework introduced in this section to define or

program the QoS metrics of importance to specific applications. Consider, for example, a geology application that must receive the samples from a remote seismic sensor no later than 5 ms after the value is sampled. That is, the transmission delay must never be higher than 5 ms. In such case, the following QoS metric cannot exceed 0.005 for any window $[b_w, e_w]$:

$$peak\_delay^*: T \times T \times \Phi \rightarrow R$$

$$peak\_delay^*(b_w, e_w, \Pi_S) = \max_{k \text{ in } \Pi_S[b_w \le s \le e_w]}\{a_k - s_k\}.$$

QoS metrics that involve more than one stream are an extension of the metrics for a single stream. In this case, $qos^*: T \times T \times \Phi^n \rightarrow R$ where n is the number of streams involved.

For example, to assure lip synchronization in a video conference, applications must guarantee that the audio and the video streams are synchronized. In a scenario in which five audio messages carry the sound for a single video message, the *rate* in the audio channel must be five times the *rate* of the video channel. In addition, it would be desirable to have precisely five audio messages coming between the arrival of two consecutive video ones. Let Π_A and Π_V be the audio and video stream profiles, respectively. The QoS metric to measure synchronization between these streams can be defined as:

$$audio\_rate\_sync^*: T \times T \times \Phi^2 \rightarrow R$$

$$audio\_rate\_sync^*(b_w, e_w, \Pi_A, \Pi_V) = \begin{cases} 1, \text{ if } \forall\ k, k +_a 1 \text{ in } \Pi_V, \\ \qquad\qquad \left|\Pi_A[a_k \le a \le a_{k+_a 1}]\right| = 5 \\ 0, \text{otherwise.} \end{cases}$$

Video and audio are synchronized if and only if $audio\_rate\_sync^*(b_w, e_w, \Pi_A, \Pi_V) = 1$.

A.4 What Is a QoS Violation?

A QoS violation occurs when a QoS metric is outside of a tolerance interval. A QoS violation function defines in reality a class of functions, *violation*, where each function measures violation of a specific metric. Given a QoS metric *qos*, $violation_{qos}$ indicates whether a stream violates *qos*. Formally:

$violation_{qos}: \Phi \times R \times R \rightarrow \{T, F\}$

$violation_{qos}(\Pi_S, v_{\min}, v_{\max}) = F$, if and only if $v_{\min} \leq qos(\Pi_S) \leq v_{\max}$.

Similarly to the study of QoS metrics, violations can be approximated and checked over windows of time. Formally:

$violation_{qos}^*: T \times T \times \Phi \times R \times R \rightarrow \{T, F\}$

$violation_{qos}^*(b_w, e_w, \Pi_S, v_{\min}, v_{\max}) = F$, if and only if

$v_{\min} \leq qos^*(b_w, e_w, \Pi_S) \leq v_{\max}$.

Consider, for example, the monitoring of *rate* on a stream. QoS metric $violation_{rate}^*(b_w, e_w, \Pi_S, v_{\min}, v_{\max}) = F$ whenever the average rate delivered by the network in the window $[b_w, e_w]$ is within the interval $[v_{\min}, v_{\max}]$.

A.5 Filters Control QoS Performance

Management mechanisms, whether activated by applications or by a network management system, need to control QoS delivery. It is desirable for such control to be accomplished through uniform mechanisms rather than ad-hoc tricks.

A *filter* is simply an operator on a message stream that modifies its performance pro-

file. A *filter* is defined as $\varphi: \Phi \rightarrow \Phi$. $\varphi(\Pi)$ is a transformation of Π.

For example, $\varphi(\{\langle s, a, p\rangle\} \cup \Pi) = \{\langle s + 0.005, a, p\rangle\} \cup \varphi(\Pi)$ is a filter that delays the messages of a stream by delaying the sending time of each message by 5 ms.

A.6 QuAL Implements the Model

The proposed model is implemented in the QuAL language, as explained in Chapter 2. QuAL time-stamps individual messages m_k with their s_k, a_k, and p_k values. These can be used to define any metric using the C language constructs. The universal metrics are pre-defined in QuAL.

Appendix B

An Overview of Concert/C

The first design of QuAL is an extension of the Concert/C language. This appendix discusses Concert/C in greater details.

Concert [Yemini et al. 89, Auerbach et al. 91] is a family of language extensions to support distributed computing using the *process model* [Hoare 78]. In the process model, *processes* are units of execution that communicate and synchronize with one another through message-passing.

In the approach used by Concert, the process model is supported directly within a new language by adding extensions to the language. These extensions integrate in a language the concepts of processes and ports, and a set of operations. A process is mapped into any active entity that performs computations. Ports are communication end-points that can be further classified as inports, or outports, depending on whether they are receiving or sending messages, respectively. An inport contains a queue to store messages that arrive and cannot be processed immediately (a process receives messages in their order of arrival). An outport contains a *binding*, that is, a capability of placing messages on an inport queue. The type of a port is defined by the type of the messages it sends or receives, and only ports of the same type can be bound.

The set of operations integrated in a language enables the creation and termination of

processes, creation and termination of bindings, and the actual communication between processes. Any language in the Concert family will support these operators. Two forms of *Inter-Process Communication* (*IPC*) are supported: *asynchronous* and *synchronous* message passing. In the asynchronous mode, a process sends a message and continues executing. The receiving process can dequeue the message after it arrives. The synchronous mode is equivalent to a RPC [Nelson 81, Soares 92]. RPCs in the Concert model are made transparently through function pointers. The interface for performing procedure calls through function pointers is the same for both local and remote calls. In the remote case, however, the function pointer contains a binding that points to the remote function. The process making the remote call blocks until the process receiving the call executes the function and returns the results.

Concert/C [Auerbach 92] is a new language that extends C [Kernighan and Ritchie 88] to support distributed computing according to the approach defined by Concert. A Concert/C process is an executing C program. Concert/C introduces input ports as a new data type. Ports can be declared as being functions that can be called from another process (*functionports*), or simply as plain ports (*receiveports*). Functionports are defined by adding the keyword *port* to a function declaration. Receiveports are declared as follows:

---

receiveport {<message_type>} <identifier>

The clause <*message_type*> identifies the type of messages received through the port identified by <*identifier*>.

A binding is simply a pointer to an input port. A binding can reference any port of the

type it points to. When a process is created, the parent process obtains an initialized binding (pointer) to an input port in the child process that can be used by the parent to initiate communications with the child. This input port of the child process is known as *initial-port*. The keyword *initial* is used to identify the respective initial port in a child process.

Two operators are provided to check whether there are enqueued messages on a particular inport. Operator *select* accepts a list of ports as arguments and blocks until at least one of the ports has a message. The value returned by this operator is the index (that is, the position of the port in the argument list) of a busy port. Operator *poll* is a non-blocking version of *select*, which returns the value 0 if all the queues are empty. The *receive* operator is used to dequeue a message from an inport. If the queue is empty, *receive* blocks until a message is enqueued. A process that defines a functionport uses the operator *accept* to receive a message sent to this port, execute the function associated to it, and return the results to the sender. *accept* accepts a list of functionports and waits until at least one of them has a message. It then receives a message representing a function call (called a *callmessage*) from a non-empty functionport, invokes the associated function with parameters supplied from the message, and returns results from the function invocation to the calling process.

The operator *send* supports asynchronous message passing, returning as soon as the underlying system supporting Concert/C has copied the message to its internal buffer.

Concert/C supports process management, that is, creation and termination. Suppose that a program has been compiled by the Concert/C compiler and stored in the file *serv_sql4* on the machine *cs.columbia.edu*. Another process can create, and later termi-

nate this process as follows:

```
[[  program server "serv_sql4"
    newspace host "cs.columbia.edu";   ]]

main()
{ ...
    server_handle = create(server, &imit_port);
  ...
  terminate(server_handle);
}
```

The declaration inside the double brackets consists of a *distributed linking declaration*, that is, a declaration that controls the instantiation and linking of distributed programs. In the example, the declaration defines the variable *server* of type *prog_t*. The type *prog_t* stores a program description that can be instantiated into a running process using operator *create*. The variable will contain the object code from file *serv_sq14*. The *create* operator stores in the memory position designated by its second argument (the address of the variable *init_port*) the binding (pointer) to the initialport of the process created. The *create* operator returns a reference to the child process created (stored in the variable *server_handle*). The operator *terminate* terminates a process. It accepts as argument the process reference returned by the *create* operator.

Appendix C

Syntax and Informal Semantics of QuAL

This appendix describes the syntax and informal semantics of QuAL language constructs. Chapter 2 presented examples of how these constructs can be used to manage QoS performance.

This appendix is organized as follows. QuAL supports handling of two types of QoS metrics: resource level and applications specific QoS metrics. Section C.1 and Section C.2 describe QuAL constructs for the specification of resource level QoS metrics and their monitoring. Section C.1 concentrates on metrics associated with communications, whereas Section C.2 concentrates on metrics associated with computations. Section C.3 describes constructs for the monitoring of application specific metrics. Section C.4 discusses the specification of filters, as defined in Appendix A. Section C.5 presents QuAL operators to access communication temporal properties (e.g., sending and arriving time of messages). Section C.6 discusses QuAL operators to dynamically re-negotiate QoS. Finally, Section C.7 presents QuAL operators to access QoS MIB objects.

When defining the syntax, the following convention is used. Keywords and constructs from QuAL are written in ***bold*** face, from Concert/C are <u>*underlined*</u>, and from C are *plain text*.

C.1 Handling of Resource Level QoS Metrics for Communications

In QuAL, QoS measures are part of the specification of the type of a port. QuAL extends the Concert/C port type into the real time port type. The general form for the specification of real time ports is as follows:

<*real-time-port*> ::= ***realtm*** [***{<real-time-port-type-attributes>}***]
<*handlers*> <*concert-port-definition*>;

<*handlers*> :: = [***handlers {<list-of-handlers>}***]

<*concert-port-definition*> ::= <*concert-outport-definition*>
<*concert-inport-definition*>

The keyword ***realtm*** (short for ***real time***) classifies the port as a real time port and causes the QuAL runtime to maintain the temporal properties (sending, arriving, and processing times) of the messages communicated through them. QoS metrics are specified in the **<*real-time-port-type-attributes*>** clause. The **<*handlers*>** clause contains the specification of the exception handler ports associated with the port being specified. While QoS attributes are part of the type of a real time port, the declaration of exception handler ports is only a specifier of the port being defined and are not part of its type. As defined in Concert/C, the <*concert-port-definition*> clause specifies the type of the messages exchanged through a port and indicates whether the port is an inport or an outport. Real time ports are completely backwards compatible with Concert/C ports, supporting all access operations defined for them.

The general form for the specification of real time port attributes is as follows:

<***real-time-port-type-attributes***> ::= <***res-qos***> <***list-of-filters***>

The clause <***res-qos***> contains the specification of resource level QoS metrics, while <***list-of-filters***> contains the specification of filters, as defined in Appendix A. Section C.4 discusses the specification of filters in greater detail.

The general form for the specification of <***res-qos***> attributes is as follows:

<***res-qos***> ::= [<***loss***>***;***] [<***permutation***>***;***]
[<***rate***> [***,*** <***window***>]***;***] [<***peak***> [***,*** <***window***>]***;***]
[<***delay***> [***,*** <***window***>]***;***] [<***jitter***> [***,*** <***window***>]***;***]
[<***recovery***> [***,*** <***window***>]***;***]
[***multiple*** <***integer***> [***,*** ***combined***]***;***]
[***multicast;***]

<***loss***> ::= ***noloss*** [***,*** ***nocoercion***] | ***loss NULL*** [***,*** <***window***>] |
loss <***constant-expression***> [***,*** <***window***>] [***,*** ***nocoercion***]
<***permutation***> ::= ***nopermt*** [***,*** ***nocoercion***] | ***permt NULL*** [***,*** <***window***>] |
permt [***,*** <***window***>] [***,*** ***nocoercion***]
<***rate***> ::= ***rate NULL*** | ***rate*** <***range***> [***,*** ***nocoercion***]
<***peak***> ::= ***peak NULL*** | ***rate*** <***rate-expression***> [***,*** ***nocoercion***]
<***delay***> ::= ***delay NULL*** | ***delay*** <***time-expression***> [***,*** ***nocoercion***]
<***jitter***> ::= ***jitter NULL*** | ***jitter*** <***time-expression***> [***,*** ***nocoercion***]
<***recovery***> ::= ***recovery NULL*** | ***recovery*** <***time-expression***> [***,*** ***nocoercion***]

<***window***> ::= ***window*** <***time-expression***>
<***range***> ::= <***time-expression***> ***-*** <***time-expression***> | <***time-expression***> ***-*** |
- <***time-expression***>
<***rate-expression***> ::= <***time-expression***>
<***time-expression***> ::= <***unit***> <***constant-expression***>
<***unit***> ::= ***ms*** | ***sec*** | ***min*** | ***hr***

The specification of any of these attributes is optional. When an attribute is not speci-

fied, the port receives the respective QoS type of a communication in Concert/C. That is, no tolerance for loss or permutation and unbounded rate, peak, delay, jitter, and recovery time intervals. The keywords ***noloss*** and ***nopermt*** indicate that loss and permutation, respectively, are not tolerated. The keyword ***NULL*** indicates that the attribute can assume any value. The range of values for this attribute is coerced if a port is bound to another one with more restrict constraints. The coercion mechanism used in QuAL was discussed Chapter 2. The keyword ***nocoercion*** indicates that coercion is not allowed for a metric. The clause **<*window*>** indicates how often QuAL runtime must measure a metric. The keyword ***multiple*** indicates if an inport can be connected to more than one outport at a time. In this case, the keyword ***combined*** indicates if the rate and peak QoS constraints apply to all connections combined. When this keyword is omitted, the rate and peak constraints apply to each connection individually.

QuAL automatically monitors QoS performance on communications and signal applications when QoS violations are detected. QuAL signals violations by sending exception messages to application specified exception handler ports, as explained in Chapter 2. Exception handler ports associated with a port are specified through the **<*handlers*>** clause as shown below. The clause **<*filter-handler*>** is discussed in Section C.4.

<*handlers*> :: = [***handlers {<list-of-handlers>}***]

<*list-of-handlers*> ::= **<*res-handler*>** **<*filter-handler*>**

<*res-handler*> ::= ***res_handler*** *<port-reference>****;***

C.2 Handling of Resource Level QoS Metrics for Computations

QuAL supports the specification of resource level QoS metrics associated with computations through real time blocks. The general form for a real time block is defined as follows:

<real-time-block> :: =
within (<timing-constraint-expression>)
do {<timed-block>}
[***miss_deadline {<timed-instruction-list>}***]
until(<condition>)

The semantics are as follows. The QuAL runtime analyzes the current system load every time an application reaches a ***<real-time-block>***. As a result of this analysis, the runtime system decides whether there are enough processing resources available for the execution of the ***<timed-block>*** without violating the timing constraints specified in ***<timing-constraint-expression>***. If there are not enough resources, control is passed to the statement following the ***<real-time-block>***. Otherwise, the application enters the real time mode, in which the ***<timed-block>*** is executed according to ***<timing-constraint-expression>***, until ***<condition>*** evaluates to a positive value. If the timing constraints in ***<timing-constraint-expression>*** are ever violated, control is passed to the ***<timed-instruction-list>*** following the keyword ***miss_deadline***. The clause ***<timed-instruction-list>*** will be elaborated later in this section.

Whenever control reaches the statement following ***<real-time-block>***, the global variable ***qual_status*** indicates the reason for leaving the real time execution. It indicates if

there were not enough resources available (with a value less than zero), or if **<*condition*>** evaluated to a positive value (with a value greater than or equal to zero). In the last case, the value of ***qual_status*** indicates how many times the **<*timed-block*>** was executed to completion before **<*condition*>** evaluated to a positive value.

The timing constraints of a **<*real-time-block*>** are defined as follows:

<*timing-constraint-expression*> ::= **<*behavior*>** [**<*start*>**]

<*behavior*> ::= ***hard; <hard-schedule>*** | ***soft; <soft-schedule>***
<*start*> ::= ***after (<event-list>)***

<*event-list*> ::= **<*event*>** [|| **<*at-event-list*>**]
<*hard-schedule*> ::= ***periodic; <period>*** | ***sporadic; <period>; <at-event-list>;***
<*soft-schedule*> ::= **<*hard-schedule*>*;*** | ***aperiodic; <at-event-list>;***

<*event*> ::= *select(<port>)* | **<*event-variable-identifier*>**
<*at-event-list*> ::= ***atEvent (<event-list>)***
<*period*> ::= ***period <rate-expression>;***

<*event-variable-identifier*> ::= *<identifier>*

When the **<*start*>** clause is specified, the first execution of the **<*timed-block*>** only happens after one of the events defined in this clause occurs. When not specified, the first execution can happen at any time. The events recognized by QuAL are the arrival of a message, which can be tested through the Concert/C `select` operator, or the change of state of a condition variable. The execution pattern of applications with ***soft*** or ***hard*** constraints can be periodic or sporadic. In addition, application with ***soft*** constraints can also have an aperiodic execution pattern. Periodic applications are those that must be processed at regular intervals, and must be completed before the next instance is due. These

regular intervals are defined by <***period***>. Sporadic applications are asynchronous activities triggered by one of the events in <***at-event-list***>. These events, however, do not occur more often than the rate indicated in <***period***>. Aperiodic applications are also asynchronous activities triggered by events. However, it can not be predicted how often these events will occur. It is important to note the finer granularity for the expression of real time constraints in QuAL, when compared to other real time languages [Halang and Stoyenko 91]. Constraints are expressed per set of instructions (blocks), rather than per application. Therefore, a QuAL application may have several blocks of code each with a different behavior.

The general form for a <***timed-block***> is as follows:

<***timed-block***> ::= <***timed-instruction***> [<***timed-block***>] |
<***timeout-block***> [<***timed-block***>]

<***timeout-block***> ::= ***timeout(***[<***time-expression***>]***) {***<***list-of-instructions***>***}***
<***expired-handler***>

<***list-of-instructions***> ::= <***instruction***> [<***list-of-instructions*** >]
<***expired-handler***> ::= ***expired {***<***timed-instruction-list***>***}***

<***timed-instruction-list***> ::= <***timed-instruction*** > [<***timed-instruction-list***>]

A <***timed-block***> consists of a sequence of <***timed-instruction***>s and <***timeout-block***>s. A <***timed-instruction***> is any instruction for which the computational cost can be determined at compile time. The set of <***timed-instruction***>s supported depends on the version of the QuAL compiler used and must be checked accordingly. An instruction or sequence of instructions for which the computational cost cannot be estimated has to be

inside a <***timeout-block***>. At compile time, the computational cost of these instructions is estimated to be equal to <***time-expression***>. During runtime time, if the execution lasts longer than <***time-expression***>, control is passed to <***expired-handler***>. When <***time-expression***> is not specified, the QuAL compiler estimates (infers) a time value for the execution of the block.

C.3 Handling of Application Specific QoS Metrics

QuAL runtime automates monitoring of application specific QoS metrics. This section first describes how application developers can trigger the monitoring of application specific metrics. It then shows how applications can instruct the QuAL runtime to signal violations to the metrics defined.

QuAL includes the operator ***qual_monitor*** to enable monitoring of application specific metrics and the operator ***qual_terminate_monitoring*** to cancel it. The general form for these operators is defined as follows:

<metric-id> qual_monitor (<qos-metric-function-identifier>, <window-time>, <number-of-ports>, <list-port-values>)

<metric-id> ::= *<integer>*
<qos-metric-function-identifier> ::= *<function-identifier>*
<window-time> ::= *<integer>*
<number-of-ports> ::= *<integer>*
<list-port-values> ::= ***<port-value>*** [***, <port-value>***]

<qual-status> qual_terminate_monitoring (<metric-id>)

<qual-status> ::= *<integer>*

The operator ***qual_monitor*** causes the runtime to perform two main tasks. First, the runtime will generate the performance profile (as defined in Appendix A) of all the ports listed in <***list-port-values***>. The parameter <***number-of-ports***> indicates how many ports there are in <***list-port-values***>. Second, the runtime will evaluate the function <***qos-metric-function-identifier***> every interval of <***window-time***> seconds, passing the performance profiles of the ports in <***list-port-values***> as its arguments. The operator <***qos-metric-function-identifier***> is a function that given a set of performance profiles it returns the value of an application specific QoS metric. Application developers are responsible for designing these functions. The runtime stores the values returned by <***qos-metric-function-identifier***> into QoS MIB objects, as explained in Chapter 4. The operator ***qual_monitor*** returns a <***metric-id***> that can be used to cancel the monitoring.

The operator ***qual_terminate_monitoring*** causes the runtime to cancel monitoring of a QoS metric. The argument <***metric-id***> identifies the monitoring to be canceled. The operator ***qual_terminate_monitoring*** returns a <***qual-status***> value indicating if the monitoring was successfully canceled.

The signature of functions in a <***qos-metric-function-identifier***> clause must be of the following type:

---

double (*) (double, double, ***qos_ppp*** **, ...)*

The function returns a value of type *double* that indicates the value of the QoS metric it measures. The function accepts the time when the window started, the time when the window ended, and one *performance profile pointer* for each port being measured. The total

number of arguments the function takes depends on the number of ports being measured. Performance profile pointers are defined as follows:

```
typedef struct pps {
  double ts;
  double ta;
  double tp;
} qos_pps;

typedef struct ppp {
  int size;
  qos_pps* signatures[];
} qos_ppp;
```

A performance profile pointer (***qos_ppp***) contains the number of performance signatures in the profile (field *size*) and a vector of performance signatures (*signatures*). Each performance signature is of type ***qos_pps***. A signature indicates the connection the message belongs to (*conn*) and the time the message was sent (field *ts*), arrived (*ta*), and was processed (*tp*). A connection is identified by the port that is sending the messages (*origin*) and by the port that is receiving them (*target*). In QuAL, variables that indicate time are of type *double* because they store values of the sysUpTime object [Stallings 93] maintained by the local management system. This object measure the number of milliseconds since the system was last initialized.

QuAL introduces the operator ***qual_violation_signalling*** to trigger QoS violation signaling and the operator ***qual_terminate_violation_signalling*** to cancel it. The general for of these operators is as follows:

```
<metric-id> qual_violation_signalling (<qos-metric-function-identifier>,
                <window-time>, <min-value>, <max-value>, <port-handler>,
```

<number-of-ports>, <list-port-values>)

<*min-value*> ::= *<integer>*
<*max-value*> ::= *<integer>*
<*port-handler*> ::= *<port-value>*

<*qual-status*> *qual_terminate_violation_signalling (<metric-id>)*

Similarly to the operator ***qual_monitor***, the operator ***qual_violation_signalling*** causes the runtime to generate the performance profile of the ports in **<*list-port-values*>** and to evaluate **<*qos-metric-function-identifier*>** every interval of **<*window-time*>**. In addition, however, it sends a message to the port **<*port-handler*>** if the value returned by **<*qos-metric-function-identifier*>** is not greater than **<*min-value*>** or lower than **<*max-value*>**. The message includes **<*metric-id*>**, the time the measure was done, and the value returned by the function. Applications use ***qual_terminate_violation_signalling*** to cancel a QoS violation signaling.

C.4 Specifying Filters

QuAL provides the means for the specification of filters, as defined in Appendix A. Filters are defined per port and are part of the **<*real-time-port-type-attributes*>**, as discussed in Section C.1. The general form for the specification of filters is as follows. QuAL supports two types of filters: single stream filters and multiple stream filters. Single stream filters define constraints that are related to a single stream (specified in a **<*single-stream-filter*>** clause), while multiple stream filters define constraints related to a group of streams (specified in a **<*multiple-stream-filter*>** clause). For each type of filter, program-

mers can specify the filters a port is able to comply with and the filters a port demands that connecting ports comply with. The first set of filters is specified after the keywords ***cmpl*** and ***grp_cmpl***. The second set is specified after the keywords ***conn_cmpl*** and ***grp_conn_cmpl***. It is important to notice that the filter identification for a group stream filter is an enumeration identifier. Each element of the enumeration represents one of the streams in the group. The declaration of a group filter for a port must specify which stream that port represents. This is accomplished by specifying the *<enum-element-identifier>* inside the brackets.

<***real-time-port-type-attributes***> ::= <***res-qos***> <***list-of-filters***>

<***list-of-filters***> ::= <***single-stream-filter***> [<***list-of-filters*** >] |
    <***multiple-stream-filter***> [<***list-of-filters*** >]

<***single-stream-filter***> ::= ***cmpl {<list-of-single-filter-identifications>}*** |
    conn_cmpl {<list-of-filter-identifications>}
<***multiple-stream-filter***> ::= ***grp_cmpl {<list-of-group-filter-identifications>}*** |
    grp_conn_cmpl {<list-of-group-filter-identifications>}

<***list-of-single-filter-identifications***> ::= <***single-filter-identifier***>;
    [<***list-of-single-filter-identifications***>]
<***list-of-group-filter-identifications***> ::=
    <***group-filter-member*** >; [<***list-of-group-filter-identifications***>]

<***single-filter-identifier***> ::= *<identifier>*
<***group-filter-member*** > ::=
    <group-filter-identifier>[<group-filter-member-identifier>]

<***group-filter-identifier***> ::= *<enum-identifier>*
<***group-filter-member-identifier***> ::= *<enum-identifier>*

Application developers must bind to each filter identifier a filter function that will actually implement the filtering. QuAL introduces the operator ***assg*** to associate a filter

identifier to a filter function. The general form for this operator is defined as follows:

<qual-status> assg (<port-value>, <filter-reference>,
<filter-function-identifier>)

<filter-reference> ::= ***<single-filter-identifier>*** | ***<group-filter-identifier>***
<filter-function-identifier> ::= *<function-identifier>*

The operator ***assg*** associates the filter function ***<filter-function-identifier>*** to the filter ***<filter-reference>*** for the port ***<port-value>***. The signature of single stream filter functions for outports must be of the following type:

---

mon_t *(*) (int index, double sending_time)*

The first argument is an index that identifies the order of the message in the communication according to its sending time. The second argument indicates the time in which the message was sent. Filter functions return values that indicate whether a message complies with the constraints defined by the filter. These functions return a value of type ***mon_t*** defined as follows:

```
typedef struct {
  int exception;
  int remove;
} mon_t;
```

A positive value in the field *remove* indicates that the message must be filtered, whereas a positive value in the field *exception* indicates that an exception must be raised.

The signature of single stream filter functions for inports must be of the following type:

***mon_t** (*) (int index, double sending_time, double arriving_time)*

The value returned and the first two arguments have semantics similar to filter functions for inports. The third argument, however, indicates the time the message arrived.

The signature of multiple stream filter functions for inports must be of the following type:

***mon_t** (*) (enum filter_member, int index, double sending_time)*

The only difference between the signature of a single stream filter function and a multiple stream function is the additional first argument that indicates the stream that generated the message. Similarly, the signature for multiple stream filter functions for inports must be of the following type:

***mon_t** (*) (enum filter_member, int index, double sending_time, double arriving_time)*

Messages filtered from a communication and messages indicating exceptions are sent to application specified exception handler ports. These handler ports are specified in the **<*handlers*>** clause of a port declaration, defined as follows:

<*handlers*> :: = [***handlers {*<*list-of-handlers*>*}***]

<*list-of-handlers*> ::= **<*res-handler*>** **<*filter-handler*>**

<filter-handler> ::= ***fil_handler {<list-of-filter-port-references>};***

<list-of-filter-port-references> ::=
    <single-filter-identifier> *<port-reference>;* [***<list-of-filter-port-references>***] |
    <group-filter-identifier> *<port-reference>;* [***<list-of-filter-port-references>***]

Each element in this clause indicates a filter and the respective handler port.

C.5 Accessing Communication Temporal Properties

QuAL introduces a set of operators that allow the retrieval of temporal properties of messages, that is, their sending, arriving, and processing times. The general form of these operators is as follows:

<qual-status> receive_tm (*<port-value>****, ****<message-ref>****,***
    <time-ref>, <time-ref>, <time-ref>)
<qual-status> rtm_receive (<time>, <time>,
    *<port-value>****, ****<message-ref>****)***
<qual-status> rtm_receive_tm (<time>, <time>,
    *<port-value>****, ****<message-ref>****,***
    <time-ref>, <time-ref>, <time-ref>)

<time-ref> ::= *<assignment-expression>*
<time> ::= *<time-value>*

The operator ***receive_tm*** causes a message to be dequeued from the port designated by *<port-value>*, and be placed on the storage designated by *<message-ref>*. It also places the message sending time, arrival time, and retrieval time on the storage designated

by the first, second, and third ***<time-ref>*** arguments, respectively. If no message is present, it blocks waiting for the next message. The operator ***rtm_receive*** causes a message that arrived in the time interval defined by the first and second arguments to be dequeued from the port designated by *<port-value>* and be placed on the storage designated by *<message-ref>*. The operator ***rtm_receive_tm*** operates similarly, but, in addition, they place the message's sending time, arrival time, and retrieval time on the storage designated by the last three ***<time-ref>*** arguments, respectively. If there are no messages that arrived in the time interval specified, all these operations will block until either a message arrives or the time constraints are unreachable. In the last case, the operations will return an error value, signaling the exception.

C.6 Re-negotiating QoS Metrics Dynamically

QuAL provides two operators to enable dynamic QoS re-negotiation. The operator ***qos_get*** is used to retrieve the QoS negotiated for a communication. The operator ***re_negotiate*** causes the runtime to re-negotiate QoS for a communication. These operators are defined as follows.

---

re_negotiate (<*port-value*>***) {<res-qos>}***

---

qos_get (<*port-value*>***) {<real-time-port-attribute-references>}***

<***real-time-port-attribute-references***> ::=
    <***res-qos-reference***> <***list-of-filter-references***>

<***res-qos-reference***> ::=
    [***loss*** <*integer-reference*> [***; window*** <***time-reference***>]***;***]
    [***permt*** <*boolean-reference*> [***; window*** <***time-reference***>]***;***]
    [***rate*** <***range-reference***> [***; window*** <***time-reference***>]***;***]
    [***peak*** <***rate-expression-reference***> [***; window*** <***time-reference***>]***;***]
    [***delay*** <>]
    [***jitter*** <*integer-reference*>]
    [***recovery*** <*integer-reference*>]
    [***multiple*** <*integer-reference*>]
    [***combined*** <*boolean-reference*>]

<***rate***> ::= ***rate NULL*** | ***rate*** <***range***> [***, nocoercion***]
<***peak***> ::= ***peak NULL*** | ***rate*** <***rate-expression***> [***, nocoercion***]
<***delay***> ::= ***delay NULL*** | ***delay*** <***time-expression***> [***, nocoercion***]
<***jitter***> ::= ***jitter NULL*** | ***jitter*** <***time-expression***> [***, nocoercion***]
<***recovery***> ::= ***recovery NULL*** | ***recovery*** <***time-expression***> [***, nocoercion***]

<***window***> ::= ***window*** <***time-expression***>
<***range***> ::= <***time-expression***> ***-*** <***time-expression***> | <***time-expression***> ***-*** |
    - <***time-expression***>
<***rate-expression***> ::= <***time-expression***>
<***time-expression***> ::= <***unit***> <***constant-expression***>
<***unit***> ::= ***ms*** | ***sec*** | ***min*** | ***hr***

C.7 Accessing QoS MIB Objects

QuAL includes operators that enable applications to access QoS MIB instrumentation directly, bypassing SNMP agents. The general form of these operators is as follows. These operators return a pointer to the memory position where the value retrieved from the QoS MIB is stored. The operator ***snmp_get*** retrieves the value of <***object-identifier***>. The ar-

gument **<*object-identifier*>** uniquely identifies a QoS MIB according to SNMP standards. The operator ***qual_app_get*** returns the value of the instance of **<*application-table-object-name*>** associated with the application **<*application-identifier*>**. Similarly, the operators ***qual_in_get*** and ***qual_out_get*** return the values of the instances of **<*inport-table-object-name*>** and **<*outport-table-object-name*>**, respectively, associated with the ports identified by their first argument.

---

*(void *)* ***snmp_get (<object-identifier>)***
*(void *)* ***qual_app_get (<application-identifier>, <application-table-object-name>)***
*(void *)* ***qual_in_get (***<*port-reference*>***, <inport-table-object-name>)***
*(void *)* ***qual_out_get (***<*port-reference*>***, <outport-table-object-name>)***

<*object-identifier*> ::= <*identifier*> | <*identifier*>.<*identifier*>
<*application-identifier*> ::= <*process-id*>
<*inport-table-object-name*>*)* ::= <*identifier*>
<*outport-table-object-name*>*)* ::= <*identifier*>

Appendix D

QoS MIB Definition

This appendix contains the *Structure of Management Information* (*SMI*) [Case et al. 93] definition of the QoS MIB. QoS MIB objects belong to one of the following groups:

- *Application* (qApp for short), described in Section D.1,
- *Outport* (qOut), described in Section D.2,
- *Inport* (qIn), described in Section D.3 and
- *Programmable* (qProg), described in Section D.4.

D.1 Application Group

The application group consists of the qAppTable table which will have one row for each QuAL application running on the system. The only static information held on the application is its name. All other static information should be obtained from the directory service supported by the system. The qAppDirectoryName is an external key, which allows an qAppTable entry to be cleanly related to an X.500 Directory [Halsall 92] entry.

```
QuAL-MIB DEFINITIONS ::= BEGIN
IMPORTS
    OBJECT-TYPE, Experimental, Counter32
            FROM SNMPV2-SMI
    DisplayString, TimeStamp
            FROM SNMPV2-TC
```

```
-- This MIB Module uses the extended OBJECT-TYPE macro as
-- defined in RFC 1212.

qApplication MODULE-IDENTITY
    LAST-UPDATED "9510170000Z"
    ORGANIZATION "Columbia University - DCC Laboratory"
    CONTACT-INFO
            "               Patricia Florissi
             Postal:        Columbia University
                            Computer Science Department
                            500 West 120th Street - Room 450
                            NYc, NY             10027
                            USA
            E-Mail:         pgsf@cs.columbia.edu"
    DESCRIPTION
            "The MIB module describing applications written in QuAL."
    ::= {experimental 47}

-- The basic qAppTable contains a list of the application entities.
qAppTable OBJECT-TYPE
    SYNTAX SEQUENCE OF QAppEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION
            "The table holding objects which apply to all QuAL applications."
    ::= {qApplication 1}

qAppEntry OBJECT-TYPE
    SYNTAX QAppEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION
            "An entry associated with a QuAL application. A QuAL
            application is a executing entity that was written in QuAL.
            From an OS point of view, an OS-level process can consist
            of a single heavy-weight QuAL application or several light-
            weight QuAL applications."
    INDEX {qAppOSProcess, qAppIndex}
    ::= {qAppTable 1}

QAppEntry ::= SEQUENCE {
    qAppOSProcess
            INTEGER,
    qAppId
```

INTEGER,
qAppName
DistinguishedName,
qAppDirectoryName
DistinguishedName,
qAppUpTm
TimeStamp,
qAppOperStatus
INTEGER,
qAppLstChng
TimeStamp,
qAppPeriod
INTEGER,
qAppCost
INTEGER,
qAppAccSoft
Counter32,
qAppAccHard
Counter32,
qAppAccSoftTm
INTEGER,
qAppAccHardTm
INTEGER
qAppLstSoft
TimeStamp,
qAppLstHard
TimeStamp,
qAppMissDead
Counter32,
qAppLstSchdFail
TimeStamp,
qAppExpTmout
Counter32,
qAppLstExpTmout
TimeStamp,
qAppLstSoftFail
TimeStamp,
qAppLstHardFail
TimeStamp,
qAppManager
DisplayString,
qAppMgtStatus
DisplayString
qAppInCnn
Counter32,

```
        qAppOutCnn
                Counter32,
        qAppAccInCnn
                Counter32,
        qAppAccOutCnn
                Counter32,
        qAppLstInCnn
                TimeStamp,
        qAppLstOutCnn
                TimeStamp,
        qAppLstInCnnFail
                TimeStamp,
        qAppLstOutCnnFail
                TimeStamp,
        qAppInCnnFail
                Counter32,
        qAppOutCnnFail
                Counter32
    }

    qAppOSProcess OBJECT-TYPE
        SYNTAX INTEGER (1..2147483647)
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "An index (the OS process number) to uniquely identify the
                OS-level process running on a host machine."
        ::= {qAppEntry 1}

    qAppIndex OBJECT-TYPE
        SYNTAX INTEGER (1..2147483647)
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "An index to uniquely identify a QuAL light weight process
                inside an OS process running on a host machine."
        ::= {qAppEntry 2}

    qAppName OBJECT-TYPE
        SYNTAX DistinguishedName
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "The name the QuAL application chooses to be known by.
                That is, the name of the file that contains the executable."
```

```
    ::= {qAppEntry 3}

qAppDirectoryName OBJECT-TYPE
    SYNTAX DistinguishedName
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The Distinguished Name of the directory entry where static
            information about this application is stored. An empty string
            indicates that no information about the application is avail-
            able in the directory."
    ::= {qAppEntry 4}

qAppUpTm OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The value of sysUpTime at the time the QuAL application
            was initialized. If the application was initialized prior to the
            last initialization of the network management subsystem,
            then this object contains a zero value."
    ::= {qAppEntry 5}

qAppOperStatus OBJECT-TYPE
    SYNTAX INTEGER {
            “nrt” (1),
            “hp” (2),
            “hs” (3),
            “sp” (4),
            “ss” (5),
            “sa” (6),
            “down” (7)
    }
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Indicates the operational status of the QuAL application. A
            QuAL application can be executing in non real time mode
            (“nrt”), or in the real time mode. In the last case, the first
            letter indicates the mode (hard or soft), and the second let-
            ter the behavior (periodic, sporadic, or aperiodic). The value
            “down” indicates that the application is not available (e.g., it
            is in the zombie state) or has died.”
    ::= {qAppEntry 6}
```

```
qAppLstChng OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The value of sysUpTime at the time the QuAL application
            entered its current operational state (e.g., nrt, hp, etc.). If
            the current state was entered prior to the last initialization of
            the local network management subsystem, then this object
            contains a zero value."
    ::= {qAppEntry 7}

qAppPeriod OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Number of times per second that the real time computation
            should be scheduled."
    ::= {qAppEntry 8}

qAppCost OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Estimated computational cost in milliseconds of the real
            time compuation in execution, if any."
    ::= {qAppEntry 9}

qAppAccSoft OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Total number of times the application executed in soft real
            time mode."
    ::= {qAppEntry 10}

qAppAccHard OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Total number of times the application executed in hard real
```

```
            time mode."
    ::= {qAppEntry 11}

qAppAccSoftTm OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Total amount of time the application executed in soft real
            time mode."
    ::= {qAppEntry 12}

qAppAccHardTm OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Total amount of time the application executed in hard real
            time mode."
    ::= {qAppEntry 13}

qAppLstSoft OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Last time the application executed in soft real time mode."
    ::= {qAppEntry 14}

qAppLstHard OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Last time the application executed in hard real time mode."
    ::= {qAppEntry 15}

qAppMissDead OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Number of times the application missed a deadline."
    ::= {qAppEntry 16}
```

```
qAppLstSchdFail OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Last time when a deadline was missed."
    ::= {qAppEntry 17}

qAppExpTmout OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Number of times the timeout for executing a real time task
            expired."
    ::= {qAppEntry 18}

qAppLstExpTmout OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Last time when a timeout expired."
    ::= {qAppEntry 19}

qAppLstSoftFail OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Last time when a request to execute in soft real time mode
            was rejected due to lack of processing resources available."
    ::= {qAppEntry 20}

qAppLstHardFail OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Last time when a request to execute in hard real time mode
            was rejected due to lack of processing resources available."
    ::= {qAppEntry 21}

qAppManager OBJECT-TYPE
    SYNTAX DisplayString
```

```
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Entity currently managing QoS violations."
    ::= {qAppEntry 22}

qAppMgtStatus OBJECT-TYPE
    SYNTAX DisplayString
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "QoS violation control request from an application to an
            SNMP manager or vice versa."
    ::= {qAppEntry 23}

qAppInCnn OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The total number of open connections to QoS demanding
            inports."
    ::= {qAppEntry 24}

qAppOutCnn OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The total number of open connections to QoS demanding
            outports."
    ::= {qAppEntry 25}

qAppAccInCnn OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The total number of connections opened to QoS demand-
            ing inports, since the application initialized."
    ::= {qAppEntry 26}

qAppAccOutCnn OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
```

```
        STATUS current
        DESCRIPTION
                "The total number of connections opened to QoS demand-
                ing outports, since the application initialized."
        ::= {qAppEntry 27}

    qAppLstInCnn OBJECT-TYPE
        SYNTAX TimeStamp
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "Time when the last connection to a QoS demanding inport
                was established."
        ::= {qAppEntry 28}

    qAppLstOutCnn OBJECT-TYPE
        SYNTAX TimeStamp
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "Time when the last connection to a QoS demanding out-
                port was established."
        ::= {qAppEntry 29}

    qAppLstInCnnFail OBJECT-TYPE
        SYNTAX TimeStamp
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "Time when the last connection to a QoS demanding inport
                was rejected."
        ::= {qAppEntry 30}

    qAppLstOutCnnFail OBJECT-TYPE
        SYNTAX TimeStamp
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "Time when the last connection to a QoS demanding out-
                port was rejected."
        ::= {qAppEntry 31}

    -- The basic qOutTable contains a list of the outport entities.
    -- It is defined in Section D.2.
    qOutTable OBJECT-TYPE
```

```
    ...

-- The basic qInTable contains a list of the inport entities.
-- It is defined in Section D.3.
qInTable OBJECT-TYPE
    ...

-- The basic qProgTable contains a list of the application programmed QoS metrics.
-- It is defined in Section D.4.
qProgTable OBJECT-TYPE
    ...
END
```

D.2 Outport Group

The outport group consists of the qOutTable table which will have one row for each outport of QuAL applications running on the system.

```
QuAL-MIB DEFINITIONS ::= BEGIN
-- . Definitions from Section D.1.
...
-- The basic qOutTable contains a list of the outport entities.
-- It augments the information in the qAppTable with information about
-- QoS demanding outports.
qOutTable OBJECT-TYPE
    SYNTAX SEQUENCE OF QOutEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION
        "The table holding information on QoS demanding outports
        of applications in qAppTable."
    ::= {qApplication 2}

qOutEntry OBJECT-TYPE
    SYNTAX QOutEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION
        "An entry associated with a QoS demanding outport of a
        QuAL application."
    INDEX {qOutLocalAdd, qOutTransPort, qRemoteAdd, qOutRemoteTransPort}
    ::= {qOutTable 1}

QOutEntry ::= SEQUENCE {
```

```
qOutOSProcessIndex
        INTEGER,
qOutApplIndex
        INTEGER,
qOutQuALPortId
        INTEGER (1..2147483647),
qOutLocalAdd
        OBJECT IDENTIFIER,
qOutTransPort
        INTEGER,
qOutRemoteHost
        DisplayString,
qOutRemoteOSProcessIndex
        INTEGER (1..2147483647),
qOutRemoteAppIndex
        INTEGER (1..2147483647),
qOutRemoteQuALPortId
        INTEGER (1..2147483647),
qOutRemoteAdd
        OBJECT IDENTIFIER,
qOutRemoteTransPort
        INTEGER,
qOutStatus
        INTEGER,
qOutEstTime
        TimeStamp,
qOutActTime
        TimeStamp,
qOutProtocol
        OBJECT IDENTIFIER,
qOutLoss
        INTEGER,
qOutPermut
        INTEGER,
qOutMinRate
        INTEGER,
qOutMaxRate
        INTEGER,
qOutPeak
        INTEGER,
qOutDelay
        INTEGER,
qOutInterDelay
        INTEGER,
qOutRecTime
```

```
            INTEGER,
    qOutMsgSize
            INTEGER,
    qOutManager
            OBJECT IDENTIFIER,
    qOutMsgSent
            TimeStamp,
    qOutCnnFail
            Counter32,
    qOutLstFail
            TimeStamp,
    qOutAccRecTime
            INTEGER,
    qOutVolume
            Counter32,
    qOutLstMsg
            TimeStamp
}

qOutOSProcessIndex OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index (the OS process number) to uniquely identify the
            OS-level process that owns the outport."
    ::= {qOutEntry 1}

qOutApplIndex OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to uniquely identify a QuAL light weight process
            inside qOutOSProcessIndex.”
    ::= {qOutEntry 2}

qOutQuALPortId OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "A number to uniquely identify a QuAL outport”.
    ::= {qOutEntry 3}
```

```
qOutLocalAdd OBJECT-TYPE
    SYNTAX OBJECT IDENTIFIER
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to uniquely identify the network address of the
            machine where qOutProcessIndex is executing. For IP
            based environments, it consists of the IP address of the ma-
            chine”.
    ::= {qOutEntry 4}

qOutTransPort OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to uniquely identify the transport-layer port allo-
            cated for qOutQuALPortId”.
    ::= {qOutEntry 5}

qOutRemoteHost OBJECT-TYPE
    SYNTAX DisplayString
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The name of the host system where the application con-
            nected to qOutOSProcess is running. For an IP based envi-
            ronment, this should be either a domain name or an IP ad-
            dress. For an OSI application it should be the string en-
            coded distinguished name of the managed object. For
            X.400(84) MTAs which do not have a Distinguished Name,
            the RFC1327 syntax 'mta in globalid' should be used."
    ::= {qOutEntry 6}

qOutRemoteOSProcessIndex OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            “A number (the OS process number) to uniquely identify the
            OS-level process connected to qOutProcessIndex.”
    ::= {qOutEntry 7}

qOutRemoteAppIndex OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
```

```
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to uniquely identify a QuAL light weight process
            inside qOutRemoteOSProcessIndex.”
    ::= {qOutEntry 8}

qOutRemoteQuALPortId OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "A number to uniquely identify the remote QuAL inport
            connected to qOutQuALPortId”.
    ::= {qOutEntry 9}

qOutRemoteAdd OBJECT-TYPE
    SYNTAX OBJECT IDENTIFIER
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to uniquely identify the network address of the
            machine where qOutRemoteOSProcessIndex is executing.
            For IP based environments, it consists of the IP address of
            the machine”.
    ::= {qOutEntry 10}

qOutRemoteTransPort OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to uniquely identify the transport-layer port allo-
            cated for qOutRemoteQuALPortId”.
    ::= {qOutEntry 11}

qOutStatus OBJECT-TYPE
    SYNTAX INTEGER {
            “up” (1),
            “down” (2)
    }
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Indicates the operational status of the connection to
```

```
            qOutQuALPortId. A connection can either be “up” or
            “down”.”
    ::= {qOutEntry 12}

qOutEstTime OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            " The value of sysUpTime when the connection was estab-
            lished.”
    ::= {qOutEntry 13}

qOutActTime OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            " The value of sysUpTime when the traffic became active,
            that is, the first message was sent.”
    ::= {qOutEntry 14}

qOutProtocol OBJECT-TYPE
    SYNTAX OBJECT IDENTIFIER
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An identification of the protocol being used for the com-
            munication through qOutQuALPortId. Currently, the fol-
            lowing protocols are supported: ST-II, ATM Transport,
            TCP/IP, UDP/IP, Unix Local Protocols."
    ::= {qOutEntry 15}

qOutLoss OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The probabilistic message loss rate (10^(-qOutLoss)) tol-
            erated for the communication.”
    ::= {qOutEntry 16}

qOutPermut OBJECT-TYPE
    SYNTAX INTEGER {
            “no” (1),
```

```
            “yes” (2)
    }
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "It indicates if the communication tolerates permutation
            (“yes”) or not (“no”).”
    ::= {qOutEntry 17}

qOutMinRate OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Minimum number of messages per second that must be de-
            livered in the communication.”
    ::= {qOutEntry 18}

qOutMaxRate OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Maximum average number of messages per second that
            will be transmitted.”
    ::= {qOutEntry 19}

qOutPeak OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Maximum number of messages per second that will be
            transmitted during peak periods.”
    ::= {qOutEntry 20}

qOutDelay OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Maximum propagation delay tolerated for the communica-
            tion.”
    ::= {qOutEntry 21}
```

```
qOutInterDelay OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Maximum delay variance tolerated for the communica-
            tion."
    ::= {qOutEntry 22}

qOutRecTime OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Maximum time in milliseconds tolerated for recovery from
            connection failures."
    ::= {qOutEntry 23}

qOutMsgSize OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Maximum message size in number of bytes."
    ::= {qOutEntry 24}

qOutManager OBJECT-TYPE
    SYNTAX OBJECT IDENTIFIER
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Identifier of the entity currently controlling communication
            QoS violations."
    ::= {qOutEntry 25}

qOutMsgSent OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Total number of messages sent since the connection was
            established."
    ::= {qOutEntry 26}

qOutCnnFail OBJECT-TYPE
```

```
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Total number of connection failures since the connection
            was first established."
    ::= {qOutEntry 27}

qOutLstFail OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The value of the sysUpTime when the last connection
            problem occurred."
    ::= {qOutEntry 28}

qOutAccRecTime OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Total amount of time in milliseconds spent in recovering
            from failures."
    ::= {qOutEntry 29}

qOutVolume OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Total volume of data in kilobytes sent since the connection
            became active."
    ::= {qOutEntry 30}

qOutLstMsg OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Value of sysUpTime when the last message was sent."
    ::= {qOutEntry 31}

-- The basic qInTable contains a list of the inport entities.
-- It is defined in Section D.3.
```

```
qInTable OBJECT-TYPE
    ...

-- The basic qProgTable contains a list of the application programmed QoS metrics.
-- It is defined in Section D.4.
qProgTable OBJECT-TYPE
    ...
END
```

D.3 Inport Group

The inport group consists of the qInTable table which will have one row for each in-port of QuAL applications running on the system.

```
QuAL-MIB DEFINITIONS ::= BEGIN
-- . Definitions from Section D.1.
...
-- . Definitions from Section D.2.
...
-- The basic qInTable contains a list of the inport entities.
-- It augments the information in the qAppTable with information about
-- QoS demanding inports.
qInTable OBJECT-TYPE
    SYNTAX SEQUENCE OF QInEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION
        "The table holding information on QoS demanding inports
        of applications in qAppTable."
    ::= {qApplication 3}

qInEntry OBJECT-TYPE
    SYNTAX QInEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION
        "An entry associated with a QoS demanding inport of a
        QuAL application."
    INDEX {qInLocalAdd, qInTransPort, qRemoteAdd, qInRemoteTransPort}
    ::= {qInTable 1}

QInEntry ::= SEQUENCE {
    qInOSProcessIndex
```

INTEGER,
qInApplIndex
INTEGER,
qInQuALPortId
INTEGER (1..2147483647),
qInLocalAdd
OBJECT IDENTIFIER,
qInTransPort
INTEGER,
qInRemoteHost
DisplayString,
qInRemoteOSProcessIndex
INTEGER (1..2147483647),
qInRemoteAppIndex
INTEGER (1..2147483647),
qInRemoteQuALPortId
INTEGER (1..2147483647),
qInRemoteAdd
OBJECT IDENTIFIER,
qInRemoteTransPort
INTEGER,
qInStatus
INTEGER,
qInEstTime
TimeStamp,
qInActTime
TimeStamp,
qInProtocol
OBJECT IDENTIFIER,
qInLoss
INTEGER,
qInPermut
INTEGER,
qInMinRate
INTEGER,
qInMaxRate
INTEGER,
qInPeak
INTEGER,
qInDelay
INTEGER,
qInInterDelay
INTEGER,
qInRecTime
INTEGER,

```
        qInMsgSize
                INTEGER,
        qInManager
                OBJECT IDENTIFIER,
        qInCnnFail
                Counter32,
        qInLstFail
                TimeStamp,
        qInAccRecTime
                INTEGER,
        qInMsgIndex
                INTEGER,
        qInLstMsg
                TimeStamp,
        qInMsgCounter
                INTEGER,
        qInMsgVolume
                Counter32,
        qInAccDelay
                Counter32,
        qInAccJitter
                Counter32,
        qInMsgOutOrderIndex
                INTEGER,
        qInLstOutOrderMsg
                TimeStamp,
        qInMsgOutOrderCounter
                INTEGER,
        qInMsgOutOrderVolume
                Counter32,
        qInAccOutOrderDelay
                Counter32,
        qInAccOutOrderJitter
                Counter32,
        qInMsgProcIndex
                INTEGER,
        qInLstProcMsg
                TimeStamp,
        qInMsgProcCounter
                INTEGER,
        qInMsgProcVolume
                Counter32
    }

    qInOSProcessIndex OBJECT-TYPE
```

```
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index (the OS process number) to uniquely identify the
            OS-level process that owns the inport."
    ::= {qInEntry 1}

qInApplIndex OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to uniquely identify a QuAL light weight process
            inside qInOSProcessIndex."
    ::= {qInEntry 2}

qInQuALPortId OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "A number to uniquely identify a QuAL inport".
    ::= {qInEntry 3}

qInLocalAdd OBJECT-TYPE
    SYNTAX OBJECT IDENTIFIER
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to uniquely identify the network address of the
            machine where qInProcessIndex is executing. For IP based
            environments, it consists of the IP address of the machine".
    ::= {qInEntry 4}

qInTransPort OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to uniquely identify the transport-layer port allo-
            cated for qInQuALPortId".
    ::= {qInEntry 5}

qInRemoteHost OBJECT-TYPE
```

```
        SYNTAX DisplayString
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "The name of the host system where the application con-
                nected to qInOSProcess is running. For an IP based envi-
                ronment, this should be either a domain name or an IP ad-
                dress. For an OSI application it should be the string en-
                coded distinguished name of the managed object. For
                X.400(84) MTAs which do not have a Distinguished Name,
                the RFC1327 syntax 'mta in globalid' should be used."
        ::= {qInEntry 6}

    qInRemoteOSProcessIndex OBJECT-TYPE
        SYNTAX INTEGER (1..2147483647)
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                “A number (the OS process number) to uniquely identify the
                OS-level process connected to qInProcessIndex.”
        ::= {qInEntry 7}

    qInRemoteAppIndex OBJECT-TYPE
        SYNTAX INTEGER (1..2147483647)
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "An index to uniquely identify a QuAL light weight process
                inside qInRemoteOSProcessIndex.”
        ::= {qInEntry 8}

    qInRemoteQuALPortId OBJECT-TYPE
        SYNTAX INTEGER (1..2147483647)
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "A number to uniquely identify the remote QuAL outport
                connected to qInQuALPortId”.
        ::= {qInEntry 9}

    qInRemoteAdd OBJECT-TYPE
        SYNTAX OBJECT IDENTIFIER
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
```

```
            "An index to uniquely identify the network address of the
            machine where qInRemoteOSProcessIndex is executing.
            For IP based environments, it consists of the IP address of
            the machine".
    ::= {qInEntry 10}

qInRemoteTransPort OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to uniquely identify the transport-layer port allo-
            cated for qInRemoteQuALPortId".
    ::= {qInEntry 11}

qInStatus OBJECT-TYPE
    SYNTAX INTEGER {
            "up" (1),
            "down" (2)
    }
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Indicates the operational status of the connection to qIn-
            QuALPortId. A connection can either be "up" or "down"."
    ::= {qInEntry 12}

qInEstTime OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            " The value of sysUpTime when the connection was estab-
            lished."
    ::= {qInEntry 13}

qInActTime OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            " The value of sysUpTime when the traffic became active,
            that is, the first message was received."
    ::= {qInEntry 14}
```

```
qInProtocol OBJECT-TYPE
    SYNTAX OBJECT IDENTIFIER
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An identification of the protocol being used for the com-
            munication through qInQuALPortId. Currently, the follow-
            ing protocols are supported: ST-II, ATM Transport,
            TCP/IP, UDP/IP, Unix Local Protocols."
    ::= {qInEntry 15}

qInLoss OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The probabilistic message loss rate (10^(-qInLoss)) toler-
            ated for the communication."
    ::= {qInEntry 16}

qInPermut OBJECT-TYPE
    SYNTAX INTEGER {
            "no" (1),
            "yes" (2)
    }
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "It indicates if the communication tolerates permutation
            ("yes") or not ("no")."
    ::= {qInEntry 17}

qInMinRate OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Minimum number of messages per second that must be de-
            livered in the communication."
    ::= {qInEntry 18}

qInMaxRate OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
```

```
    DESCRIPTION
            "Maximum average number of messages per second that
            will be transmitted."
    ::= {qInEntry 19}

qInPeak OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Maximum number of messages per second that will be
            transmitted during peak periods."
    ::= {qInEntry 20}

qInDelay OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Maximum propagation delay tolerated for the communica-
            tion."
    ::= {qInEntry 21}

qInInterDelay OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Maximum delay variance tolerated for the communica-
            tion."
    ::= {qInEntry 22}

qInRecTime OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Maximum time in milliseconds tolerated for recovery from
            connection failures."
    ::= {qInEntry 23}

qInMsgSize OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
```

```
        DESCRIPTION
                "Maximum message size in number of bytes."
        ::= {qInEntry 24}

    qInManager OBJECT-TYPE
        SYNTAX OBJECT IDENTIFIER
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "Identifier of the entity currently controlling communication
                QoS violations."
        ::= {qInEntry 25}

    qInCnnFail OBJECT-TYPE
        SYNTAX Counter32
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "Total number of connection failures since the connection
                was first established."
        ::= {qInEntry 26}

    qInLstFail OBJECT-TYPE
        SYNTAX TimeStamp
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "The value of the sysUpTime when the last connection
                problem occurred."
        ::= {qInEntry 27}

    qInAccRecTime OBJECT-TYPE
        SYNTAX INTEGER (1..2147483647)
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
                "Total amount of time in milliseconds spent in recovering
                from failures."
        ::= {qInEntry 28}

    qInMsgIndex OBJECT-TYPE
        SYNTAX INTEGER (1..2147483647)
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
```

```
            "Index (according to sending time) of the last message that
            arrive in sequence."
    ::= {qInEntry 29}

qInLstMsg OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The value of sysUpTime when the last message in se-
            quence arrived."
    ::= {qInEntry 30}

qInMsgCounter OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The total number of messages that arrived in sequence."
    ::= {qInEntry 31}

qInMsgVolume OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The total volume of data in kilobytes received in se-
            quence."
    ::= {qInEntry 32}

qInAccDelay OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The sum of the propagation delay of all messages that ar-
            rived in sequence."
    ::= {qInEntry 33}

qInAccJitter OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The total sum of the inter message delay of all messages
```

```
            that arrived in sequence."
        ::= {qInEntry 34}

    qInMsgOutOrderIndex OBJECT-TYPE
        SYNTAX INTEGER (1..2147483647)
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
            "Index (according to sending time) of the last message that
            arrive out of sequence."
        ::= {qInEntry 35}

    qInLstOutOrderMsg OBJECT-TYPE
        SYNTAX TimeStamp
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
            "The value of sysUpTime when the last message arrived out
            of sequence."
        ::= {qInEntry 36}

    qInMsgOutOrderCounter OBJECT-TYPE
        SYNTAX INTEGER (1..2147483647)
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
            "The total number of messages that arrived out of se-
            quence."
        ::= {qInEntry 37}

    qInMsgOutOrderVolume OBJECT-TYPE
        SYNTAX Counter32
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
            "The total volume of data in kilobytes received out of se-
            quence."
        ::= {qInEntry 38}

    qInAccOutOrderDelay OBJECT-TYPE
        SYNTAX Counter32
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION
            "The sum of the propagation delay of all messages that ar-
```

```
            rived out of sequence."
    ::= {qInEntry 39}

qInAccOutOrderJitter OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The total sum of the inter message delay of all messages
            that arrived out of sequence."
    ::= {qInEntry 40}

qInMsgProcIndex OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "Index (according to sending time) of the last message that
            was processed."
    ::= {qInEntry 41}

qInLstProcMsg OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The value of sysUpTime when the last was processed."
    ::= {qInEntry 42}

qInMsgProcCounter OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The total number of messages that was processed."
    ::= {qInEntry 43}

qInMsgProcVolume OBJECT-TYPE
    SYNTAX Counter32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The total volume of data in processed."
    ::= {qInEntry 44}
```

```
-- The basic qProgTable contains a list of the application programmed QoS metrics.
-- It is defined in Section D.4.
qProgTable OBJECT-TYPE
    ...
END
```

D.4 Programmable Group

The programmable group consists of the qProgTable table which will have one row for each QoS metric programmed by QuAL applications running on the system.

```
QuAL-MIB DEFINITIONS ::= BEGIN
-- . Definitions from Section D.1.
...
-- . Definitions from Section D.2.
...
-- . Definitions from Section D.3.
...
-- The basic qProgTable contains a list of the application programmed QoS metrics.
-- It augments the information in the qAppTable with information about
-- application programmed QoS metrics.
qProgTable OBJECT-TYPE
    SYNTAX SEQUENCE OF QProgEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION
        "The table holding information on application programmed
        QoS metrics."
    ::= {qApplication 4}

qProgEntry OBJECT-TYPE
    SYNTAX QProgEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION
        "An entry associated with an application programmed QoS
        metric."
    INDEX {qProgInOut, qProgLocalAdd, qProgTransPort, qProgMet}
-- Note that the values of qProgLocalAdd, qProgTransPort, qProgRemoteAdd,
-- and qProgRemoteTransPort uniquely identify an entry in the tables qOutTable
-- and qInTable. These entries contain more information on the communication
-- being measured.
    ::= {qProgTable 1}
```

```
QProgEntry ::= SEQUENCE {
    qProgInOut
            INTEGER,
    qProgLocalAdd
            OBJECT IDENTIFIER,
    qProgTransPort
            INTEGER,
    qProgMet
            DisplayString,
    qProgWindow
            INTEGER,
    qProgLstTime
            TimeStamp,
    qProgVal
            INTEGER,
    qProgRemoteAdd
            OBJECT IDENTIFIER,
    qProgRemoteTransPort
            INTEGER
}

qProgInOut OBJECT-TYPE
    SYNTAX INTEGER {
            “in” (1),
            “out” (2)
    }
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to indicate if the metric is being measured on the
            inport (“in”) or on the outport (“out”) side of the communi-
            cation."
    ::= {qProgEntry 1}

qProgLocalAdd OBJECT-TYPE
    SYNTAX OBJECT IDENTIFIER
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to uniquely identify the network address of the
            machine where the application that owns the port being
            measured is executing. For IP based environments, it con-
            sists of the IP address of the machine."
    ::= {qProgEntry 2}
```

```
qProgTransPort OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to uniquely identify the transport-layer port allo-
            cated for the inport being measured".
    ::= {qProgEntry 3}

qProgMet OBJECT-TYPE
    SYNTAX DisplayString
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "An index to identify the name of the QoS metric pro-
            grammed by an application".
    ::= {qProgEntry 4}

qProgWindow OBJECT-TYPE
    SYNTAX DisplayString
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The size of window measured in milliseconds over which
            the metric is being measured".
    ::= {qProgEntry 5}

qProgLstTime OBJECT-TYPE
    SYNTAX TimeStamp
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The value of sysUpTime the last time the metric was meas-
            ured".
    ::= {qProgEntry 6}

qProgVal OBJECT-TYPE
    SYNTAX INTEGER
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The value of the QoS metric the last time it was meas-
            ured".
    ::= {qProgEntry 7}
```

```
qProgRemoteAdd OBJECT-TYPE
    SYNTAX OBJECT IDENTIFIER
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The network address of the machine where the application
            connected to the port being measured is executing. For IP
            based environments, it consists of the IP address of the ma-
            chine."
    ::= {qProgEntry 8}

qProgRemoteTransPort OBJECT-TYPE
    SYNTAX INTEGER (1..2147483647)
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
            "The transport-layer port allocated for the port connected to
            the one being measured”.
    ::= {qProgEntry 9}

END
```

Appendix E

Related Work

This appendix reviews related work and positions QoSME and its components in this context. Section E.1 presents related work in the areas addressed by QuAL: distributed computing, characterization and handling of QoS metrics, and real time programming. Section E.2 reviews existing designs to add QoS handling to sockets and OSs and relates them to QoSockets and QoSOS. Finally, Section E.3 addresses how QoSME distinguishes itself from existing QoS frameworks.

E.1 Related Work on QuAL

QuAL comprises work in three distinct areas: distributed computing, reviewed in Section E.1.1, characterization and handling of QoS, reviewed in Section E.1.2, and real-time programming, reviewed in Section E.1.3.

E.1.1 Distributed Computing

A *distributed system* is an application that must be executed on a distributed computing environment (possibly heterogeneous). The *process model* [Hoare 78] is an abstraction that reflects main objects and their relationships in a distributed system. A computation is performed by autonomous processing units which interact and collaborate to solve a particular task by exchanging messages. In this section, current approaches for programming

distributed systems in lieu of the process model are reviewed and arguments are made as to why Concert/C was chosen as our basic engine.

From an application programmer perspective, communication facilities are provided at levels as low as the transport layer [Stallings 94]. The most basic transport-layer service is the *socket* [Stevens 90] abstraction. Sockets are *Service Access Points* (*SAPs*) that entail transport-layer addresses used by processes to communicate. Sockets allow the specification of low level communication details, such as connectionless or connection-oriented service. No compile-time or run-time support is provided to verify more application-oriented features such as the type of information being exchanged, flow of control issues, etc. All these high-level features must be explicitly coded by the application developer. This requires a considerable level of expertise on transport-layer issues and results in non-reliable and complex codes. Furthermore, the topology of the processes in the application must be explicitly specified by the programmer in terms of the location and address of each process.

The RPC abstraction [Nelson 81, Soares 92] was created to convey these issues. RPC extends the procedure call mechanism to a distributed environment, whereby a process can call a procedure that belongs to another process. Interprocess communication is achieved by using the syntax and semantics of a well accepted strongly typed language abstraction. RPC also constitutes a sound basis to move existing applications to the distributed system environment, thus supporting software reusability. RPC enables *location transparency*, whereby a process can communicate with another process using language-level identifiers, function references, without knowledge of transport-layer details. Also, the communica-

tion abstraction is simple: synchronous communication with the caller blocking until the result of the RPC is returned from the callee. The type of the data being exchanged are the procedure parameters and the compiler is able to type-check the communication.

A natural extension of this trend was to develop completely new languages that are especially suitable for distributed computing. Examples include many programming paradigms such as object-oriented (Emerald [Jul et al. 88]), logic for distributed computing (Prolog [Schapiro 86]), functional (Concurrent ML [Reppy 91]), and process-oriented (HERMES [Strom et al. 91] and Ada [Ada 83]). A complete survey of programming languages paradigms for distributed computing can be found in [Bal et al. 89], and an analysis of several paradigms for process interaction in distributed programs in [Andrews 91].

A compromise was necessary between the conflicting goals of having a new distributed programming language and being compatible with existing environments. Thus, existing programming environments and languages have been extended in several dimensions to support distributed computing at a higher-level. The most common approach is to provide RPC packages [Soares 92]. However, due to language details, the programmer is often exposed to a complex programming interface and has to perform several tasks in order to bring the program to a state in which RPC can be directly used. Usually, the process owning the remote procedure must register in the RPC runtime in the transport layer, and in the associated name servers. Only then can it listen for arriving calls. The process calling the procedure must invoke transport-layer services, name servers, and other RPC implementation services to locate the remote procedure and establish a remote binding. These features undermine location, syntax and semantics transparency.

Concert [Yemini et al. 89, Auerbach et al. 91] was developed to overcome these drawbacks. Concert is in reality a family of language extensions, being developed at IBM T. J. Watson Research Laboratory, to support distributed computing. The approach taken by Concert minimizes the learning overhead and does not incur the loss of location, syntax, and semantics transparency. Languages are extended with a concise set of new types and operators that support the process model while allowing reuse of existing code. The process model is supported directly within new languages by these extensions. Thus, it eliminates the multiple abstractions of language, OS, and add-on packages, and replaces them with a single, higher-level abstraction. In such an integrated system, the application developer uses a single programming interface, ignoring the details of how this interface is supported. It is the responsibility of the language designer and implementer to transparently map the Concert interface into low-level services provided by the language run-time system and underlying OS.

Concert also introduces the notion of interoperability whereby programs written in any of the Concert family of languages interoperate with each other and also with services written using conventional RPC packages. The interoperability is done by the language run-time system, and it is transparent to the application developer. The Concert run-time system shelters heterogeneity at machine-level, OS-level, and communication-protocol-level. As a consequence, a program written in one language can communicate with another program written in a different language, and running on a different OS through Concert. A Concert program is also interoperable with non-Concert programs running other inter-process communication protocols.

Concert/C is the language extension to C [Kernighan and Ritchie 88] to support the Concert approach for distributed computing. Concert/C has been prototyped and it supports a strongly-typed message-passing mechanism and RPC to specify inter-process communication integrated into the language.

Because of these features, the Concert approach was chosen as the approach for distributed computing and Concert/C selected as the platform over which our extensions for the development of multimedia applications are built. A more detailed overview of Concert and Concert/C was presented in Appendix B.

E.1.2 QoS Handling

Inter-process communication abstractions are appropriate for traditional applications because they abstract communication delays that the programmer does not need to understand in order to perform its job. This is not the case for distributed multimedia applications because QoS parameters must be negotiated with the network and the application must be aware of periods in which the network is unable to provide them. Future communication abstractions must therefore explicitly enable specification, negotiation, and monitoring of such QoS parameters, while sheltering all communication details that are not relevant for the application. This section reviews efforts that have been geared towards providing QoS to applications.

Provision of QoS at the transport layer has been the subject of much effort recently. References [Feldemeier 93, Doeringer et al. 90] include an extensive review on the state of the art in this area. Protocols at the transport layer are developed to provide end-to-end guaranteed service across a network. It is the responsibility of the application to exchange

information among its peers to set up the QoS parameters, sometimes via another inter-process communication mechanism. The set up phase includes selecting the characteristics of the data flow between origin and target(s), identifying the SAP relevant to the specific transport protocol being used, and ensuring security, if necessary. Examples of parameters that may be specified are bandwidth, delay, reliability, and error selection policy. Data is transmitted as part of the point-to-point or point-to-multi-point connections. During connection setup, paths to the destination are selected and the necessary network resources are reserved. Mechanisms at the transport layer require extra effort from the application developer to understand communication details, similar to what happens with the socket abstraction.

An example of a transport-layer protocol for QoS provision is ST-II [Topolcic 90]. The motivation for this protocol was to enhance the Internet Protocol [Stevens 90] for the establishment of QoS-dependable communication streams. Several works followed ST-II, such as HeiTS [Hehmann et al. 91], CMTP [Wolfinger and Moran 91], TPX [Danthine et al. 92], RTP [Schulzrinne and Casner 95].

Transport-layer protocols that use ST-II have been developed to ease the establishment of communication streams in specific domains. For example, PVP [Cole 81] and NVP [Cohen 81] can be used directly by applications transmitting video and audio, respectively. PVP and NVP automatically choose the ST-II QoS parameters most appropriate for the transmission.

A more specialized transport protocol for audio and video digital communication across interconnected packet switched networks is described in [Jeffay et al. 92]. This

transport protocol restricts the services provided. The QoS parameters that can be specified are synchronization type between audio and video, number of frames played out of order, and end-to-end latency.

In addition to the complexity involved in the connection establishment and monitoring of such communications, applications that use transport-layer inter-process communication facilities are directly restricted to nodes that support the respective protocol. The *Session Reservation Protocol* (*SRP*) [Anderson et al. 90] is a step towards overcoming this problem. SRP is a session-layer resource reservation protocol for guaranteed-performance communication in the Internet and it works independently of any particular transport protocol. SRP can be used with standard protocols, such as IP [Postel 81], or with new protocols, such as ST-II. A host implementing SRP can use IP when communicating with hosts not supporting SRP. SRP uses the DASH resource model [Anderson et al. 90] to specify the reservation of *resource* (disk, CPU or network) capacity. SRP is directly responsible for reserving all network resources and can thus be viewed as a *network management protocol*. Several other reservation protocols followed SRP, such as RSVP [Braden et al. 95], RCAP [Benerjea and Mah 91], HieRAT [Volg et al. 95], and UNI 3.0 [ATM Forum 93]. References [Keshav 93] and [Kurose 93] contain a detailed overview of current efforts in this area.

Even session-layer protocols are low-level abstractions for application developers. The application-developer still needs to handle issues such as data representation across heterogeneous environments and inter-process synchronization model. In order to cope with these drawbacks, application-level support tools for specific domains were developed.

MCAM [Keller and Effelsberg 93] is an application-layer architecture for handling video streams in a heterogeneous distributed environment. A *source* is any entity reading from an input device and producing a stream and a *sink* is an entity consuming a stream. Remote connections between sources and sinks in a heterogeneous environment are accomplished through a MCAM well-defined protocol. Information is passed between a movie service user and a movie service provider using service primitives.

The functional model of the system consists of four parts. The *directory system* is used to store and access distributed movie information. The *Equipment Control System* (*ECS*) allows the integrated handling of remote devices, e.g., to perform a camera zoom, or to adjust the volume of speakers. The *Stream Provider System* (*SPS*) provides to MCAM a plain stream service. The *MCAM system* interacts with other systems to provide services to an MCAM user. All services can be accessed at the SAP of the MCAM layer. Every play operation creates a *context* that carries information about the current value of the stream parameters supported. Such parameters are reliability (error-free or non-error-free), speed (retrieval and transmission), mode (fast or slow motion), quality (compression algorithm used), section (parts of the movie), and direction (forward or backward). All these parameters except the quality parameter can be modified by the user, if the movie is stored in a file. For live transmission, however, the user can only change the reliability, mode and quality. SPS is limited to video-specific applications and parameters negotiation is asymmetric. The service user specifies the stream parameters and the provider cannot restrict or negotiate them.

QuAL provides an application-level abstraction for the negotiation, establishment and

management of QoS-dependable communications. The negotiation is completely symmetric. Both sender and receiver specify the QoS values desirable for the communication. The model guarantees that connections be opened only between senders and receivers that have matching QoS communication requirements. This is in contrast with MCAM, for instance, that the service user dictates the QoS parameters of the communication. Furthermore, QuAL provides an abstract model for the specification of QoS parameters that is independent of the communication protocol used, and also independent of the nature of the data being transmitted. This approach is general, it does not limit the domain of applications suitable to this framework, and it enables the bridging of heterogeneity at transport and session layers.

E.1.3 Real Time Language Constructs

One of the key issues in distributed multimedia application development is how to specify the QoS that should be delivered by the underlying runtime system. Many such QoS requirements relate to temporal behaviors. It is thus necessary to use real time language constructs to specify such QoS demands. This section presents a brief survey of real time language constructs.

There has been a significant amount of research done in providing language level support for specifying real time constraints. In such languages, one specifies the constraints of the tasks to be executed and the underlying system is responsible for reserving the resources, and for scheduling the tasks accordingly.

Real-time constraints are classified as either *hard* or *soft*. Hard real-time constraints have to be met, whereas soft real-time constraints may be eventually violated (some re-

covery mechanism to handle violations must be provided). To guarantee that hard real-time constraints are met, a worst case performance analysis of the tasks to be executed must be performed at compile time. The analysis determines each task workload and is used for the allocation of resources. Based on workloads and constraints, the system must find an execution schedule (scheduleability analysis). The fact that the execution time of all program segments must be known at compile time imposes severe restrictions on the language constructs that can be used as well as on the underlying OS. Examples of constructs that make such an estimate impossible are *while* loops. The underlying system must also provide an upper bound for OS calls, such as input and output operations. The scheduleability analysis problem is complex, and in some cases intractable [Garey and Johnson 75].

Real time systems differ on the real time constraints that can be specified, on the type of scheduleability analysis they employ, and on recovery mechanisms they support. A detailed survey of real time languages can be found in [Halang and Stoyenko 91].

Since real time system implementation imposes so many challenges, existing languages are customized to solve specific problems, without stretching all real time language requirements. Some examples of soft real time languages are RTL/2 [Barnes 76] for industrial process control, PEARL [Kappatsch 77] designed by a group of German researchers from research institutes and industrial firms, ILIAD [Pickett 79] designed by General Motors' Research Laboratories, and PORTAL [Nageli and Gorrengourt 79] designed to be used in system programming and real time process control. All these languages implement heuristic scheduleability analysis (thus causing deadlines to be missed) and weak exception-handling mechanism for the handling and recovery of such violations.

Ada [Ada 83], on the other hand, is a general purpose programming language that includes real-time capabilities. Ada was designed as the result of a procurement exercise by the U.S. Department of Defense and it is expected to become the most used real time language in the near future. Being a typical design-by-the-committee product, it includes just about every feature conceivable in a modern language. Ada hardly makes any provisions for scheduleability analysis and, as a result of its size and complexity, requires large amounts of run time support. The only way for expressing time dependencies in Ada is to delay the execution of tasks by specified periods. Thus, Ada programmers have to hand-tune task timing dependencies. Furthermore, it does not prevent deadlocks during shared-memory access and resources are allocated in a first-in-first-out basis. The multitasking model in Ada does not deliver predictable execution.

None of the recently developed hard real-time languages is in wide use, and are classified as experimental. Examples include TOMAL [Kieburtz and Hennessy 76], DICON [Lee 84, Lee and Gehlot 85], ORE [Donner and Jameson 88], FLEX [Lin and Natarjan 88], Real-time Mentat [Grimshaw et al. 89], RTC++ [Ishikawa et al. 90], and Real-Time Euclid [Kligerman and Stoyenko 86]. Real-time Euclid was chosen as a representative example of such languages, since it meets all the standard requirements to support hard real-time programming.

Real-time Euclid is a descendant of Concurrent Euclid [Cordy and Holt 81]. Programs can always be analyzed for guaranteed scheduleability, and use a structured exception handling mechanism. Inter-process synchronization mechanisms, such as monitors and signals, are analyzed to study the timing dependencies among processes and to prevent

deadlocks. Statements to wait on signals are extended to specify timeouts and corresponding exception handlers. Default system level exception handlers are defined for standard exceptions. Loops are restricted to iterate no more than a compile time known number of iterations. There is no provision for dynamic data structures, since it may introduce unpredictability in both time and storage requirements. No recursion is permitted, since it not possible to calculate an upper-bound on the number of times the function will be called.

The scheduleability analysis is completely performed during compilation. Real-Time Euclid processes can be either *periodic*, or *aperiodic*. Periodic processes are those which have to execute at regular intervals, and must be completed before the next interval is due. Aperiodic processes are asynchronous processes that have only soft deadlines and for which no minimum inter-execution time interval is known. Aperiodic processes can be activated by system interrupts, by other processes or by timers. This information is used by the scheduleability analyzer to decide if the entire system of processes is scheduleable and by the runtime system to properly schedule processes.

An extension of PEARL to support hard-real time programming has been proposed in [Halang and Stoyenko 91]. The main goal is to bring the good programming practices of Real-Time Euclid from the experimental to the industrial world. PEARL is still in the design phase, and it requires a specialized underlying OS to support its implementation. In order to guarantee the compliance of real time constraints, resources are allocated to handle the worst case scenario, that is, when the system has the highest work load. This pessimistic approach, although necessary, may lead to low system use, especially when the system load fluctuates highly. To overcome this problem, PEARL uses the concept of *im-*

precise results [Lin et al. 87] to provide graceful system degradation. Imprecise results are defined as the most recent partial results. This idea only works for monotonicaly improving computations. By making results of poorer quality available when the results of desirable quality cannot be obtained in time, real-time services of potentially inferior quality are provided in a timely fashion.

QuAL builds a general purpose, high level, real time language suitable for hard and soft real-time programming with predictable behavior. Similar to Real-time Euclid and extended PEARL, it provides high-level language constructs for the specification of process scheduling and control, and time constrained behavior. On the other hand, QuAL puts less restrictions on the underlying supporting system without sacrificing behavior predictability.

E.2 Extending Sockets and OSs to Support QoS

Several designs [Topolcic 90, Partridge 95, DePrycker 93] have been proposed to add QoS negotiation to the sockets interface. In fact, each transport protocol that supports QoS defines its own extension to the sockets interface. The main reason being that transport protocols differ on the QoS parameters supported and on the units of measurement used for certain parameters. For example, communication reservations in ST-II are made in terms of the following parameters: smallest packet size measured in user data bytes (LimitOnPDUBytes), lowest packet rate that can be tolerated by the sending side measured in tenths of a packet per second (LimitOnPDURate), and minimum bandwidth that can be tolerated by the sending side expressed as a product of bytes and tenths of a packet per second. On the other hand, communication resources in AAL are expressed in terms

of the following parameters: the desired and minimum peak bandwidth measured in kilobits per second(peak_bandwidth.target and peak_bandwidth.minimum), the desired and minimum bandwidth measured in kilobits per second (mean_bandwidth.target and mean_bandwidth.minimum), and the desired and minimum burst traffic measured in kilobits packet length (mean_burst.target and mean_burst.minimum). Thus, each protocol defines its own set of parameters through which QoS can be specified.

QoSockets distinguishes itself from existing extensions in the following main features:

- It is transport protocol independent. The QoSockets runtime does the translation between the QoS parameters supported by QoSockets and the parameters supported by the underlying protocol.
- It automatically generates SNMP QoS MIBs. These MIBs contain performance statistics on the QoS delivered to applications. At one hand, applications access these MIBs to analyze the QoS being delivered and to detect violations. On the other hand, SNMP managers access these MIBs to evaluate and control the performance of the delivered QoS.

Researchers [Govindan and Anderson 91, Feldmeier 93, Stankovic et al. 95, Coulson et al. 95] have also focused on how OSs can be extended to support the real time capabilities required by distributed multimedia applications. Reference [Aurrecoechea et al. 95] classifies these efforts into the following main approaches:

- modifying existing UNIX kernel to provide more predictable behavior, such as the work in [Hanko et al. 91], or
- completely re-implementing UNIX, such as the Chorus implementation described

in [Coulson et al. 95].

Chapter 3 describes how the functionality offered by QoSOS can be supported on OSs that support the POSIX standard [Posix 79], putting QoSOS in the first approach. However, using the techniques similar to the ones described in Chapter 3, QoSOS runtime could also be implemented to run on top of [Hanko et al. 91] and [Coulson et al. 95]. This way QoSOS applications can be ported across different underlying OSs.

E.3 QoS Frameworks

Only recently, the research community started addressing QoS handling through QoS architectures. A QoS architecture investigates how the communication network and the end computation systems attached to it need to be extended to support QoS in an integrated manner. This design is in contrast to previous research efforts that concentrated on QoS provision by specific components, such as admission control mechanisms that assure maximum transmission latency at the network.

Several architectures have been proposed, such as the *Extended Integrated Reference Model* (*XRM*) [Lazar 94], the TINA QoS Framework [Nilison et al. 95], and the *Quality of Service Architecture* (*QoS-A*) [Campbell et al. 94]. Reference [Aurrecoechea et al. 95] includes a detailed overview of existing QoS architectures. This section discusses only QoS-A, since it can be chosen as a representative example of such architectures.

QoS-A is a layered architecture that provides services and mechanisms for QoS assurance. QoS-A consists of six layers. At the very top, the *distributed systems platform* provides mechanisms to specify QoS in an object based environment. Below this layer, the *orchestration layer* provides jitter correction and multimedia synchronization services.

The *transport layer*, the *network layer*, the *data link layer*, and the *physical layer* support the orchestration layer by providing a range of QoS configurable QoS services. Each layer in this architecture includes three main functional components, called *planes*. The *protocol plane* offers services to establish communication between peer layers. The *QoS maintenance plane* monitors and maintains the services provided by the protocol plane on its layer. Finally, the *flow management plane* is responsible for managing connection establishment and translating QoS constraints between neighbor layers. By defining the interactions between layers, QoS-A defines a model for tight integration among the resources at each layer. These resources include data devices, OS threads, communication protocols, and networks.

QoSME distinguishes itself from these architectures in its generality. More specifically, QoSME enables application level QoS management independent of the underlying architecture or transport protocol system being used. On the other hand, existing QoS architectures impose specific constraints on the underlying architecture. For example, QoS-A applications require that the underlying system offers the services specified by the functional planes. Thus, QoSME promotes code portability and reusability. Furthermore, QoSME runtime can execute on top of QoS architectures and use the services provided by them. This feature enables QoSME applications to execute and make use of the functionality supported by QoS architectures in general, such as QoS-A.

Appendix F

QoSME 1.0 Manual Pages

Bibliography

[Ada 83] *The Programming Language Ada Reference Manual.* Spring-Verlag, Berlin-Heidelberg, New York - Tokyo, 1983.

[Anderson et al. 90] D. P. Anderson, S. Tzou, R. Wahbe, R. Govindan, and M. Andrews. Support for Continuous Media in the DASH System. In *Proceedings of the Tenth International Conference on Distributed Computing Systems,* Paris, May 1990.

[Andrews 91] G. R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, vol. 23, no. 1, pp. 49-90, March 1991.

[ATM Forum 93] ATM Forum. *ATM User-Network Interface Specifications, Version 3.* Prentice-Hall, 1993.

[Auerbach et al. 91] J. S. Auerbach, D. F. Bacon, A. P. Goldberg, G. Goldszmidt, M. T. Kennedy, A. R. Lowry, J. R. Russel, W. Silverman, R. E. Strom, D. M. Yellin, and S. A. Yemini. High-level language support for programming reliable distributed systems. In *Proceedings of the First CASCON International Conference,* Toronto, Canada, October 1991.

[Auerbach 92] J. S. Auerbach. Concert/C Specification. Technical Report, IBM T. J. Watson Research Center, Yorktown Heights, NY, November 1992.

[Aurrecoechea et al. 95] C. Aurrecoechea, A. Campbell, and L. Hauw. A Comparison of End to End QoS Architectures.. To appear in the *Multimedia Systems ACM Journal, Springer International*, 1995.

[Bal et al. 89] H. E. Bal, J. G. Steiner, and A. S. Tanembaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, vol. 21, no. 3, pp. 261-322, September 1989.

[Barnes 76] J. Barnes. *RTL/2 Design and Philosophy*. Heyden, London, 1976.

[Benerjea and Mah 91] A. Benerjea and B. Math. The Real-Time Channel Administration Protocol. In *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video,* IBM ENC, Heidelberg, Germany, November 1991.

[Braden et al. 95] R. Braden, L. Zhang, D. Estrin, S. Herzog, and S Jamin. Resource ReServation Protocol (RSVP) -- Version 1 Functional Specification. Internet Draft, draft-ietf-rsvp-spec-07, Integrated Services Working Group, 1995.

[Campbell et al. 94] A. Campbell, G. Coulson, and D. Hutchison. *ACM Computer Communications Review*, April 1994.

[Case et al. 93] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2). Internet Requests for Comments (RFC) 1442, April 1993.

[Cohen 81] D. Cohen. A Network Voice Protocol NVP-II. Technical Report, USC/Information Sciences Institute, April 1981.

[Cole 81] E. Cole. PVP - A Packet Video Protocol. Technical Report, W-Note 28, USC/Information Sciences Institute, August 1981.

[Comer and Stevens 91] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP Volume 1*. Prentice Hall, NJ, 1991.

[Coulson et al. 95] G. Coulson, A. Campbell, and P. Robin.
Design of a QoS Controlled ATM Based Communication System in Chorus.
IEEE Journal of Selected Areas in Communications (JSAC), Special Issue on ATM LANs: Implementation and Experiences with Emerging Technology, May 1995.

[Cordy and Holt 81] J. Cordy and R. Holt.
Specification of Concurrent Euclid.
Technical Report, CSRG-133, August 1981.

[Danthine et al. 92] A. Danthine, Y. Baguette, G. Leduc, and L. Leonard.
The OSI 95 Connection-Mode Transport Service - Enhanced QoS.
In *Proceedings of the 4th IFIP Conference on High Performance Networking*, Liege, Belgium, December 1992.

[DePrycker 93] M. De Prycker.
Asynchronous Transfer Mode: solution for Broadband ISDN.
Ellis Horwood Limited, Hemel Hempstead, Hertfordshire, England, Second Edition, 1993.

[Doeringer et al. 90] W. Doeringer, D. Dykeman, M. Kaiserwerth, B. Meister, H. Rudin, and R. Williamson.
A Survey of Light-weight transport Protocols for High-speed Networks.
IEEE Transactions on Communications, November 1990.

[Donner and Jameson 88] M. Donner and D. Jameson.
Language and operating systems features for real-time programming.
Computing Systems, 1988.

[Elmasri and Navathe 94] R. Elmasri and S. Navathe.
Fundamentals of Database Systems - Second Edition.
The Benjamin/Cummings Publishing Company, Inc., 1994.

[Feldemeier 93] D. Feldmeier.
A Framework of Architectural Concepts for High Speed Communication Systems.
Technical Report, Computer Communication Research Group, Bellcore, Morristown, May 1993.

[Frederick 94] R. Frederick.
Video Conference Tool (NetVideo).
Available through ftp://parcftp.xerox.com/pub/net-research/nvbin-3.3*.

[Florissi 95] P. Florissi.
QoSockets - Version 1.0.
Available through the World Wide Web at the address http://www.cs.columbia.edu/dcc or through ftp://ftp.cs.columbia.edu/pub/qual/qual.tar.Z, 1995.

[Freed and Kille 93] N. Freed and S. Kille.
Network Service Monitoring Management Information Base.
Internet Draft, 1995.

[Garey and Johnson 75] M. Garey and D. Johnson.
Complexity results for multiprocessor scheduling under resource constraints.
SIAM Journal on Computing, vol. 4, no. 4, 397 - 411, December 1975.

[Goldszmidt 95] G. Goldszmidt.
Distributed Management by Delegation.
PhD Thesis, Computer Science Department, Columbia University, New York, New York, 1995.

[Govindan and Anderson 91] R. Govindan and D. P. Anderson.
Scheduling and IPC Mechanisms for Continuous Media.
In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, Pacific Grove, California, USA, SIGOPS, vol. 25, pp 68-80, 1991.

[Grimshaw et al. 89] A. Grimshaw, A. Silberman, and J. Liu.
Real-Time Mentat programming language and architecture.
In *Proceedings of GLOBECOM 89*, 1989, pp. 1-7.

[Halang and Stoyenko 91] W. A. Halang and A. D. Stoyenko.
Constructing Predictable Real Time Systems.
Boston/Dordrecht/London: Kluwer Academic Publishers, 1991.

[Halsall 92] F. Halsall.
Data Communications, Computer Networks and Open Systems.
Addison Wesley, 1992.

[Hanko et al. 91] J. G. Hanko, E. M. Keurner, J. D. Northcutt, and A. G. Wall.
Workstation Support for Time Critical Applications.
In *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video*, Heidelberg, Springer Verlag, 1991.

[Hehmann et al. 91] D. B. Hehmann, R. G. Herrtwich, W. Schulz, T. Schuett, and R. Steinmetz.
Implementing HeiTS: Architecture and Implementation Strategy of the Heidelberg High Speed Transport System.
In *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video*, IBM ENC, Heidelberg, Germany, November 1991.

[Hoare 78] C. A. R. Hoare.
Communicating Sequential Processes.
Communications of ACM, vol. 21, no. 8, pp. 666-677, August 1978

[Hyman et al. 93] J. Hyman, A. Lazar, G. Pacifici.
A Separation Principle Between Scheduling and Admission Control for Broadband Switching.
IEEE Journal on Selected Areas in Communications, vol. 11, no. 4, pp. 605 - 616, May 1993.

[Ishikawa et al. 90] Y. Ishikawa, H. Tokuda, and C. Mercer.
Object-oriented real-time language design: Constructs for timing constraints.
Technical Report CMU-CS-90-111, Computer Science Department, Carnegie Mellon University, Pitsburgh, Pensilvania, 1981.

[Jeffay et al. 92] K. Jeffay, D. L. Stone, and F. D. Smith.
Transport and Delay Mechanism for multimedia Conference Across Packet-switched networks.
Journal of Computer Networks and ISDN Systems, 1992.

[Jul et al. 88] E. Jul, H. Levy, N. Hutchinson, and A. Black.
Fine-Grained Mobility in the Emerald System.
ACM Transactions on Computer Systems, vol. 6, no. 1, pp. 109-133, February 1988.

[Kappatsch 77] A. Kappatsch.
Full PEARL language description.
Technical Report, KFK-PDV 130, 1977.

[Kernighan and Ritchie 88] B. W. Kernighan and D. M. Ritchie.
The C Programming Language, Second Edition.
Englewood Cliffs, NJ: Prentice Hall, 1988.

[Keller and Effelsberg 93] R. Keller and W. Effelsberg.
MCAM: An Application Layer Protocol for Movie Control, Access, and Management.
In *Proceedings of the* in *First ACM International Conference on Multimedia*, Anaheim, 1993.

[Keshav 93] S. Keshav.
Report on the Workshop on Quality of Service Issues in High Speed Networks.
ACM Computer Communications Review, vol. 22, no. 1, pp. 6–15, January 1993.

[Kieburtz and Hennessy 76] R. Kieburtz and J. Hennessy.
TOMAL – A high-level programming language for microprocessor process control applications.
ACM SIGPLAN Notices, vol. 11, no. 4, pp. 127-134, April 1976.

[Kligerman and Stoyenko 86] E. Kligerman and A. Stoyenko.
Real-Time Euclid: A language for reliable real-time system.
IEEE Transactions on Software Engineering, vol. 12, no. 9, pp. 941 - 949, September 1986.

[Kumar 95] V. Kumar.
What is the MBONE.
Available through the World Wide Web at the address http://www.best.com/~prince/techinfo/mbone.html.

[Kurose 93] J. F. Kurose.
Open Issues and Challenges in Providing Quality of Service Guarantees in High Speed Networks.
ACM Computer Communications Review, vol. 23, no. 1, pp. 6–15, January 1993.

[Lazar et al. 90] A. Lazar, A. Temple, and R. Gidron.
An Introduction for Integrated Networks that Guarantees Quality of Service.
International Journal of Digital and Analog Communication Systems, vol 3, pp 229 - 238, April-June 1990.

[Lazar 94] A. Lazar.
Challenges in Multimedia Networking.
In *Proceedings of the International Hi-Tech Forum*, Osaka, Japan, February 1994.

[Lee 84] I. Lee.
A programming system for distributed real-time applications.
In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1984, pp. 18-27.

[Lee and Gehlot 85] I. Lee and V. Gehlot.
Language constructs for distributed real-time programming.
In *Proceedings of the IEEE Real-Time Systems Symposium*, , 1985, pp. 57-66.

[Leinbaugh 80] D. Leinbaugh.
Guaranteed response times in hard-real-time environment.
IEEE Transactions on Software Engineering, vol. 6, no. 1, pp. 85–91, January 1980.

[Lin et al. 87] K. Lin, S. Natarajan, and J. Liu.
Imprecise results: Utilizing partial computations in real-time systems.
In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1987.

[Lin and Natarjan 88] K. Lin and S. Natarajan.
Expressing and maintaining timing constraints in FLEX.
In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1988.

[Liu and Layland 73] C. Liu and J. Layland.
Scheduling algorithms for multiprogramming in a hard-real-time environment.
Journal of the ACM, vol. 20, 46 - 61, 1973.

[Nageli and Gorrengourt 79] H. Nageli and A. Gorrengourt.
Programming in PORTAL: An introduction.
Technical Report, 1979.

[Nelson 81] B. J. Nelson.
Remote Procedure Call.
PhD Thesis, Technical Report, Computer Science Department, Carnegie Mellon University, Pitsburgh, Pensilvania, 1981.

[Nilison et al. 95] G. Nilison, F. Dupuy, and C. Chapman.
An Overview of the Telecommunications Information Networking Architecture.
In *Proceedings of TINA'95,* Melbourne, Australia, 1995.

[O'Reilly 95] O'Reilly Publishing.
Internet User Survey.
Available through the World Wide Web at the address http://www.ora.com.

[Partridge 94] C. Partridge.
Sockets and a Quality of Service Manager.
Slides presented at the 31st IETF Meeting - Integrated Service Working Group (available through ftp://mercury.lcs.mit.edu/pub/interserv), 1994.

[Pickett 79] M. Picket.
ILIAD Reference Manual.
Computer Science Department, General Motors Research Laboratories, Warren, MI, April, 1979.

[Posix 79] Information technology - Portable Operating System Interface (POSIX).
Technical Report, 1990.

[Postel 81] J. Postel.
Internet Protocol - DARPA Internet Program Protocol Specification.
Internet Requests for Comments (RFC) 1014, September 1981.

[Reppy 91] J. H. Reppy.
CML: A higher order concurrent language.
In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 293 - 305, June 1991.

[Rose 93] M. T. Rose.
The Simple Book.
Prentice-Hall, 1993.

[Schapiro 86] E. Schapiro.
Concurrent Prolog: a progress report.
IEEE Computer, 1986.

[Schulzrinne and Casner 95] H. Schulzrinne and S. Casner.
RTP: A Transport Protocol for Real-Time Applications.
Internet Draft, draft-ietf-avt-rtp-05, 1995.

[Soares 92] P. G. Soares.
On Remote Procedure Call.
In *Proceedings of the Second CASCON International Conference*, Toronto, Canada, November 1992.

[Stallings 93] W. Stallings.
SNMP, SNMPv2, and CMIP.
Addison Wesley, 1993.

[Stankovic et al. 95] Stankovic.
Implications of Classical Scheduling Results for Real-Time Systems.
IEEE Computer, Special Issue on Scheduling and Real-Time Systems, June 1995.

[Stevens 90] W. R. Stevens.
UNIX Network Programming.
Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[Strom et al. 91] R. E. Strom, D. F. Bacon, A P. Goldberg, A. Lowry, D. M. Yellin, and S. A. Yemini.
Hermes — A Language for Distributed Computing.
Prentice-Hall, 1991.

[Sun Microsystems 87] Sun Microsystems.
XDR: External Data Representation Standard.
Internet Requests for Comments (RFC) 1014, June 1987.

[Sun Microsystems 92] Sun Microsystems.
SunOS 4.1.3 User's Reference Manual.
Sun Microsystems, Inc., 1992.

[Sun Microsystems 93] Sun Microsystems.
SunOS 5.3 Reference Manual.
Sun Microsystems, Inc., 1993.

[Sun Microsystems 94] Sun Microsystems.
SunOS 5.3 Guide to Multi-Thread Programming.
Sun Microsystems, Inc., 1994.

[Topolcic 90] C. Topolcic.
Internet Stream Protocol, Version 2 (ST-II).
Internet Requests for Comments (RFC) 1190, October, 1990.

[Vogel et al 94] A. Vodel, G. Bochmann, and J. Gecsei.
Distributed Multimedia Applications and Quality of Service - A Survey.
In *Proceedings of the CASCON International Conference,* Toronto, Canada, October 1994.

[Volg et al. 95] C. Volg, L. Wolf, R. Herrtwich, and H. Wittig.
HeiRAT - Quality of Service Management for Distributed Multimedia Systems.
Multimedia Systems Journal, November 1995.

[Wolfinger and Moran 91] B. Wolfinger and M. Moran.
A Continuous Media Data Transport Service and Protocol for Real-time Communication in High Speed Networks.
In *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video,* IBM ENC, Heidelberg, Germany, November 1991.

[Ullman 73] J. Ullman.
Polynomial complete scheduling problems.
In *Proceedings of the Fourth Symposium on OS Principles,* 1973, pp. 96 - 101.

[Yemini et al. 89] S. A. Yemini, G. S. Goldszmidt, A. D. Stoyenko, and Y. H. Wei.
Concert: A High Level Language Approach to Heterogeneous Distributed Systems.
In *Proceedings of the 9th International Conference on Distributed Computing Systems*, Newport Beach, CA, June 1989, pp. 162-171.