6-1-2001

Quality of service management for non-guaranteed networks

Tarek Madkour

The American University in Cairo

Schools of Engineering and Computer Science

QUALITY OF SERVICE MANAGEMENT FOR NON-GUARANTEED NETWORKS

A Thesis Submitted to

The Computer Science Department

in partial fulfillment of the requirements for

the degree of Master of Science

by

Tarek Madkour

B.Sc. Computer Science, AUC, Feb. 1996

Under the supervision of Dr. Amr El-Kadi

December 1999

The American University in Cairo

QUALITY OF SERVICE MANAGEMENT FOR NON-GUARANTEED NETWORKS

A Thesis Submitted by Tarek Madkour

To Department of Computer Science

December 1999

in partial fulfillment of the requirements for

The degree of Master of Science

has been approved by

Dr. Amr El-Kadi
Thesis Committee Chair / Adviser _____
Affiliation _____

Dr.
Thesis Committee Reader / examiner _____
Affiliation _____

Dr.
Thesis Committee Reader / examiner _____
Affiliation _____

_____	_____	_____	_____
Department Chair	Date	Dean	Date

TABLE OF CONTENTS

FIGURES / ILLUSTRATIONS

TABLES

ACKNOWLEDGEMENTS

I wish to extend my sincere thanks to my supervisor, Dr. Amr El-Kadi, for his guidance and support over the past three years. Despite being incredibly busy, he has always taken the time to discuss this work and pose further questions when required. His perspective of various issues and the occasional words of encouragement have been well appreciated.

I wish to express my deep gratitude to my family for their love, understanding and their unfailing support and assistance. Particularly, I want to thank my father for his constant help and his ingenious ideas. They have all shared with me the good and bad moments during the past years. I would not have been writing this today without their love and care.

In addition, I thank my friends, especially Hisham, for their constant support and motivation. They never failed to cheer me up when I was feeling down. Their friendship has always been the reason for my delight.

Finally, my appreciation goes to all of the staff of the Computer Science Department for their encouragement and friendship. I have enjoyed the challenges, camaraderie, and many other aspects of postgraduate study at the university.

ABSTRACT

The increasing dominance of multimedia communication posed new requirements for the underlying systems. Multimedia data, formally called continuous media, has time constraints that impose realtime limitations for their transmission. Certain levels of service, called Quality of Service (QoS), need to be considered when handling continuous media.

The present work utilizes QoS concepts for networks that do not have inherent QoS support. The thesis aims at verifying the possibility of having QoS-controlled communication on non-guaranteed networks. A basic QoS architecture is designed where already existing QoS concepts are adapted to work with non-guaranteed networks. The architecture provides the facilities of QoS specification, mapping, admission, maintenance, monitoring and notification. In addition, a new concept for predictive QoS admission is introduced.

The proposed architecture was verified using a prototype system. The results showed an increased percentage of continuous media that arrive on time to their receivers (goodput) with higher network loads. The increased goodput was at the expense of high network overhead.

Chapter 1 INTRODUCTION

The new requirements posed by the use of multimedia in distributed networks called for new ways of handling the new type of data that was being treated, continuous media. Handling continuous media favored the introduction of new mechanisms for dealing with its realtime aspects. Quality of Service (QoS) management stands out as a major aspect to be handled.

1.1 Continuous Media

With the advancement in computer hardware and networking technologies, there was a matching elevation in the anticipation of computer users. Traditional computer users started demanding more than merely exchanging text data and binary files. The next step in user expectations was to interchange audio and video data, which are collectively referred to as multimedia. Multimedia transfer imposed more burdens on the existing systems for two main reasons: the nature of the data being transferred, the increasing bandwidth requirements for multimedia data, and the nature of multimedia applications themselves.

Unlike binary data, multimedia has diverse requirements in terms of bandwidth, latency levels and jitter. Once multimedia transfer starts, it is expected to flow with a constant rate. This concept of constant flow rate gave rise to the concept of continuous media. One cannot expect an audio transfer to be delayed in the middle without hearing “clicks” in the stream, which renders the audio stream non-comprehendible. The same applies for the other types of multimedia data. It is this implied rate of

exhibition for continuous media that caused problems for traditional distributed systems. Conventional distributed systems were designed to deal with transmitted data in a “best-effort” fashion. This “best-effort” technique will destroy the integrity of continuous media because it does not guarantee the expected rates. This called for the evolution of new systems that can handle continuous media, whether at the application level, transport layer level, or distributed system level.

1.2 Quality of Service

The realtime requirements of continuous media imposed a limit on the level of service that a distributed system can provide to its clients. The guaranteed level of service for continuous media communication is termed *Quality of Service (QoS)*. QoS refers to certain characteristics as observed by the transport service users [Reynolds 96].

Most existing distributed systems are based on traditional communication architectures that have a narrow notion of QoS. The Internet Protocol (IP), for example, can only specify ‘high’ and ‘low’ throughput or reliability factors. The underlying network rarely honors such qualitative measures. Furthermore, existing qualitative QoS parameters are defined statically. Current needs require the ability to dynamically modify the agreed upon values through processes of negotiation and renegotiations. The need exists for more quantitative parameters to be specified and honored [Campbell 94]. Modern multimedia distributed systems should have the facility to specify and honor agreed QoS levels.

Recent network topologies provide support for QoS definition and management. Modern networks, such as ATM, allow for the reservation of QoS levels prior to channel creation. Once a QoS level is granted, the network guarantees to send the

transmitted data at the specified level. Older networks, such as Ethernet, do not have a facility for resource reservation. The network medium is shared among applications and it is left to the applications to cooperate in sharing the network in a way that maximizes network utilization and performance.

1.3 The Problem

This thesis addresses one problem that shows with QoS management: **the provision of system-level QoS on non-guaranteed networks**. The goal of this thesis is to borrow, adapt and develop mechanisms for porting the QoS concepts that were originally developed for guaranteed networks to work with non-guaranteed network platforms.

This work asserts that QoS provision should be performed at the system level rather than the application level. Leaving QoS provision to the application level has the following problems:

- complexity of user applications due to the addition of QoS management to the applications
- inability to handle multiple coexisting applications with different media types and varying QoS requirements

Providing a common interface to the applications to access QoS facilities will simplify the creation of multimedia applications. System-level QoS management also allows for maximum network utilization by enforcing system-level management of the data being transmitted. This thesis demonstrates that QoS can be provided at the system

level by providing a complete transport-level architecture that facilitates QoS management of multimedia data.

1.4 Motivation and Contribution

Providing QoS for non-guaranteed networks is essential due to the dominance of traditional network platforms that do not have inherent QoS support. Facilitating QoS management for the traditional platforms will allow for the utilization of the new concepts of multimedia to the old existing networks. It will also lay the foundations for permitting multimedia communication on heterogeneous networks that comprise a combination of both traditional and modern network platforms, such as the Internet.

This thesis provides a novel approach to QoS provision, where a complete transport-level QoS management architecture is designed to provide QoS functionality to non-QoS-aware platforms. The technique adopted is breadth coverage of all essential aspects of QoS management. Key QoS management functions are provided in an elementary form to furnish a basic, yet complete, framework for QoS management on non-guaranteed networks. One aspect of QoS management, namely admission control, is studied in depth to provide a new approach for predictive admission control on networks that do not provide QoS guarantees.

1.5 Thesis Outline

In the next chapter, a survey of current research in the field of QoS management is presented. The survey includes the current standardization efforts as well as individual research in the field of QoS management. The survey covers research performed on guaranteed and non-guaranteed networks.

Chapter 3 focuses on QoS management research for transport protocols. Focus is given on the incorporation of QoS mechanisms at the transport-level. A discussion of QoS management for the Internet Protocols is also presented.

The proposed architecture is outlined in Chapter 4. The chapter describes the approach used in design and details the design of the different components of the proposed system. The chapter also presents the implementation of the proposed system.

Chapter 5 provides the results of the testing and measurement performed on the implemented system. Results show the benefits of the proposed system and the costs of achieving those benefits.

Chapter 6 concludes the thesis and provides a summary of the proposed work. The chapter also outlines future directions for possible fields of research in the area of QoS management for non-guaranteed networks.

Chapter 2 QoS ARCHITECTURES SURVEY

2.1 QoS Basics

Quality of Service is the collective effect of service performance, which determines the degree of satisfaction of a user of the service [Hafid 96b]. The level of satisfaction for a distributed multimedia application is defined in terms of several characteristics, called QoS characteristics. The distributed multimedia system manages those characteristics for different application in the process of QoS management. This section describes different existing QoS characteristics and the steps of QoS management in a distributed multimedia system.

2.1.1 QoS Characteristics

QoS characteristics define the fundamental aspects of QoS to be managed by the distributed multimedia system [ISO 95]. They represent some aspects of the system that are to be identified and quantified. QoS characteristics are represented for every connection existing in a multimedia system. The QoS characteristics and the values associated with those characteristics will be assumed to exist by the application as long as the communication channel exists. In the case of multi-peer communication, the communication channel is from one sender to several recipients. Here the QoS characteristics may either be connection-wide or receiver-selected. Connection-wide characteristics apply to all instances of the communication channel with all the recipients of the service. Receiver-selected characteristics apply only to a single instance of the channel that the receiver is connected to. All other instances of the

communication channel share the connection-wide characteristics. This permits custom-configuring QoS requirements per receiver to allow for heterogeneous receivers.

The proposed ISO standard QoS framework [ISO 95] categorizes the QoS characteristics of general importance into:

- Time-related characteristics
- Coherence characteristics
- Capacity-related characteristics
- Integrity-related characteristics
- Safety-related characteristics
- Cost-related characteristics
- Security-related characteristics
- Reliability-related characteristics
- Other characteristics

Each category of the characteristics contains a set of generic QoS characteristics that could be quantified in numbers, vectors, or matrices using a specific unit. The generic QoS characteristics can be further specialized into specific QoS characteristics. Specialization is done by limiting a generic characteristic to a certain event or to or from a specific origin or location or by representing the original characteristic as a

statistical function, such as variance of the generic function. For example, the “time delay” characteristic can be specialized into “transit delay” and “request/reply” delay, which are two kinds of time delay or can be specialized into mean time delay to represent the arithmetic mean. The ISO document [ISO 95] gives a detailed discussion of the QoS characteristics in every category, their definition, quantification and units.

2.1.2 QoS Management

Distributed Multimedia Systems aim at managing QoS characteristics to produce the expected multimedia performance. QoS management is done through several QoS management functions (QMFs). QoS management functions refer to all the activities relating to the control and administration of QoS within a system. QMFs are composed of several QoS mechanisms. A QoS mechanism is an action performed by one or more entities in a distributed system to meet one or more QoS requirements. QoS mechanisms may operate individually or be combined to cooperate in performing a single QoS management function. The QoS requirements that QMFs act to meet are represented as QoS parameters, which are values given by the users of the system for certain QoS characteristics.

The activities supported by QMFs include [ISO 95]:

- Establishment of QoS for a set of QoS characteristics
- Monitoring of the observed values of QoS
- Maintenance of the actual QoS as close as possible to the target QoS
- Control of QoS targets
- Inquiry upon some QoS information or action
- Alerts as a result of some event relating to QoS management.

An important notion to QoS management is the notion of a flow. A flow is defined as the production, transmission and eventual consumption of a single media stream as an integrated activity governed by a single statement of QoS [Hutchison 95]. QoS management uses QMFs at different stages during the lifetime of a flow. QoS management is typically performed a priori, before initiation, at initiation of the flow and during interaction. A priori management can occur when certain QoS parameters are preset in the system at design time. For example, a multimedia system may have preset values for certain video characteristics that are applied system-wide. QoS management can be performed before initiation by reserving some resources before communication is initiated. During flow initiation, the application can negotiate with the system the QoS parameters to use for certain QoS characteristics according to the application needs and the system's current state. QoS management can also be performed during communication within a flow when a certain QoS characteristic falls below or exceeds the agreed upon parameters. This causes an alert to the application to either renegotiate QoS or terminate.

QoS management activities can be categorized into three phases: prediction, establishment and operation. During the prediction phase, current QoS parameters of the system are examined to be able to predict what kind of parameters an application can ask of the system. After QoS parameters are predicted, flow establishment takes place with all the QoS management functions related to establishment taking place, too. These are typically actions of negotiation, renegotiations and setting of parameters in case of degradation. Finally, during the operation of a flow, QoS monitoring and maintenance is done by the system and QoS inquiries are done by the applications. It is hence useful to categorize QoS management activities into

specification, mapping, negotiation, resource reservation, admission control, maintenance, monitoring, policing, adaptation, renegotiations, accounting and termination activities.

2.1.2.1 Specification

The purpose of QoS specification is for applications to represent their multimedia QoS requirements through defining values for the QoS characteristics supported by the systems. This is known as passing parameters to the QoS system. It is generally desirable that the system allows applications to specify their QoS requirements in terms of characteristics that are meaningful to the application. Video applications, for example, should specify the number of desired frames per second and it is up to the system to translate that (in the mapping step) to meaningful system-level QoS parameters such as throughput and jitter. The QoS requirements specified by the application will act as a service-contract that the system will be expected to adhere to.

The application can specify QoS requirements in several ways. Common ways include:

- upper or lower limits
- upper or lower thresholds
- a specific operating target value

QoS requirements can also specify actions to be taken as a result of reaching the specified limits or thresholds in other characteristics. Thresholds are different from limits in that they carry no restriction on whether or not they should be crossed [ISO 95].

2.1.2.2 Mapping

QoS mapping is normally responsible for translating the user-level QoS requirements into QoS requirements for the different levels of the system. Hafid [Hafid 96b] defines three main types for mapping:

- *QoS – QoS mapping*. QoS parameters specified at the higher level are mapped into QoS parameters for the lower layer. An example of this is mapping a protocol level time requirement into an ATM cell QoS parameter.

- *QoS – resource mapping*. QoS parameters of a certain level are mapped into resources that need to be reserved, such CPU, bandwidth or system buffers.

- *Service – system mapping*. Services are mapped onto system components that are required to support the requested service.

Mapping, in conjunction with specification, relieve the application from the burden of having to specify system-level QoS parameters that might not be meaningful to the application developer.

2.1.2.3 Negotiation

The role of the negotiation phase of QoS management is to find an agreement on the values of QoS parameters between the application establishing a flow and the distributed multimedia system. Typically, applications will ask for the best QoS they can get. It is the task of the system to check the available resources and report to the requesting application the levels of QoS that the system can permit. The application should decide, based on the reported available QoS, whether to abort establishment,

establish with a lower QoS, or negotiate a new set of parameters based on the reported QoS. For example, a video application may require 30 frames per second for its 24-bit video data. When the system reports back the availability of a 20 frames per second rate only. The application can either refuse to work with this rate, accept the 20 frames per second rate, or negotiate having 30 frames per second for 8-bit video data. The result of the negotiation process should be the establishment of the flow with acceptable QoS parameters for both the system and the application, or the cancellation of the process if agreed levels cannot be reached.

2.1.2.4 Resource Reservation

In order to guarantee the agreed QoS levels, the system needs to allocate low-level resources to the applications. Resources include network bandwidth, CPU cycles required for multimedia processing, thread scheduling, memory and buffer space. Normally, only requesting a low-level resource and checking the outcome of the request can test QoS availability. This shows why resource reservation is a process that is often tightly coupled with QoS negotiation.

Hafid [Hafid 97] discusses two approaches to resource reservation:

- *Pessimistic Approach*. Resources are allocated based on the worst-case scenario. This clearly leads to fully guaranteed QoS but also leads to under-utilization of the allocated resources because multimedia data naturally come in bursts.

- *Optimistic Approach*. Resources are allocated on the average characteristics. This causes a more efficient utilization for resources but

could yield to situations where the “guaranteed” QoS is not guaranteed! This leads us to other mechanisms of QoS alerting and QoS renegotiations.

2.1.2.5 Admission Control

Together with negotiation and resource reservation, admission control lies at the heart of QoS establishment in QoS management. Based on the system’s QoS policies and the current QoS levels, the system makes the decision on whether to admit the QoS request or not. The decision is based on tests that the system makes internally to allocate resources (resource reservation) and to inquiries about QoS levels of the requested QoS parameters. If the QoS parameters can be met without threatening QoS guarantees the QoS request is admitted. If the QoS parameters cannot be met, system policies are checked to see whether the request should be admitted. For example, an application’s request for 20% of the CPU cycles can be met even if it is not attainable if the requesting application is realtime and the system’s policies are for preempting normal applications for realtime ones. QoS renegotiations would now need to occur with the lower priority applications.

2.1.2.6 Maintenance

Since QoS management is an activity that involves the reservation of resources, distributed multimedia systems are expected to dynamically manage the reserved resources to make sure that they operate in a way that attains all contracted QoS levels. Resources are dynamically multiplexed by the system to ensure that high priority, realtime for instance, applications are guaranteed their levels and that lower priority applications do not starve. The dynamic multiplexing of resources to achieve the guaranteed QoS levels is called QoS maintenance. An example of this is the

realtime scheduler that is asked to cooperate with the QoS system to provide the required CPU cycles to the requesting applications. The maintenance of the realtime scheduler is an action of QoS maintenance.

2.1.2.7 Monitoring

QoS monitoring is another dynamic QoS management activity. It is part of the operation phase of flow management. During QoS monitoring, higher system layers constantly supervise the lower layers to ensure proper QoS levels are kept. This process involves QoS inquiry to inquire about current QoS levels and also involves comparison with the contracts to guarantee that the contracted QoS values are obeyed. Fine-grained resource adjustment can be performed as a maintenance action that results from monitoring. When contracted QoS levels cross the contracted limits, notifications are sent to the concerned applications. A QoS degradation notification is sent to applications when the QoS values they contracted cannot be maintained by the system. Applications are expected to respond to degradation notifications by ignoring them, renegotiating new levels or terminating. When QoS values exceed the high limits specified by the application (if any), another notification is sent to the application notifying it with the change. Applications can respond with renegotiations for higher QoS values or continue to run with the current levels.

Consider the video application example once more. Assume that during negotiation the application agrees on 24 frames minimum and 30 frames per second maximum for its 16 bits per pixel video. Now during operation of the flow, the system cannot keep the minimum QoS level of 24 fps. The system sends a degradation notification and the application has the option to terminate, ignore or renegotiate. The system decides to

negotiate 24-fps minimum and 30 fps maximum for black and white video. The system agrees with this and after a while, the burst in the system ceases to exist and the system can now provide more than 30 fps for black and white video for this flow. The system sends another notification to the application where it has the option to either continue with the current level or renegotiate a higher level, 30-fps 16-bit video again, for instance.

2.1.2.8 Policing

QoS policing is the equivalent of QoS monitoring but from the applications' side. Since QoS contracts are constrains to both the system and the application, QoS policing is responsible for ensuring that the application that agreed to a QoS contract does adhere to its QoS terms just as QoS monitoring ensures that the system obeys the contract, too. QoS policing is only valid where administrative or charging rules are being enforced. For example, if applications are being charged for the data they transmit and an application agrees to send at 1 MBPS; if the application tries to send at more than 1 MBPS, QoS policing should detect the situation and react. Actions expected of QoS policing include ignoring the application's violation, notifications, automatically shaping the data in the flow through filters to adhere to the contract or pure termination of the violating application.

2.1.2.9 Adaptation

QoS adaptation is responsible for automatically altering data in a flow to adapt to changing QoS context. When QoS values cannot be maintained at the contracted levels, the system may solve the problem in one of two ways: system and application policies or adaptation. The first alternative is the normal QoS alerting as a result of

QoS monitoring resulting in either renegotiations or termination. Here the application is the sole controller in case of the system being incapable of delivering the desired QoS. Another approach is to allow the system to shape the data streams to automatically adhere to QoS requirements. This philosophy is based on the concept that a degraded service is better than no service, but it ignores the fact that applications themselves know better how to degrade the service than the system. Favorers of adaptation suggest methods of data filtering to shape streams. The system may perform lossy knowledgeable compression to the data, may drop frames out of video streams or may ignore the high-quality information in scalable multimedia documents.

2.1.2.10 Renegotiation

Renegotiation is the process of repeating the QoS specification and QoS negotiation phases. It is frequently the outcome of a QoS degradation notification to the application. Other purposes for renegotiations are also commonplace. Applications may wish to save system QoS by degrading their services when they do not need the already agreed high QoS levels. Applications also tend to use the same flow for several uses, which may need different QoS levels. An example for this could be the use of a multimedia flow in a music radio broadcasting application. The application could transmit both the commentator's voice and the songs in the same stream. Naturally, commentator's voice would do with QoS levels for telephone-quality audio whereas songs would preferably be transmitted at CD-quality audio if possible. Renegotiations would have to occur every time the switch from commentator to song, or vice versa, occurs.

2.1.2.11 Accounting

Accounting activities are essential when cost is taken into account. Multimedia channels are normally attached with costs. The costs range from copyright costs of the material being transmitted to QoS-related costs of system resource usage. The QoS-related costs can be calculated in the QoS accounting phase. QoS-related costs are typically costs associated with the QoS values contracted, service guarantee types, duration of service, amount of data exchanged and security level [Hafid 96b].

2.1.2.12 Termination

QoS termination is concerned with the graceful termination of applications having QoS contracts. During termination, the system should ensure that all resources allocated to the application are released and all system structures created to support the flow are freed. Checking should also be performed to check the dependencies among applications. Applications requesting termination should be checked to ensure that all flows created by the application are not closed before the clients of those flows have ended their contract for the flow.

2.1.3 QoS Architecture Concept

QoS contracts cannot be guaranteed with the provision of QoS at a single layer of the distributed multimedia system. Early systems in QoS literature were only concerned with provision of QoS at the network and transport layers. This lead to an incomplete discussion of QoS provision. A more global look into the QoS management process as an end-to-end process needed to be considered. Later work in QoS research was aimed at integrating QoS-driven end system architecture with the network configurable QoS services and protocols in order to meet application-to-application requirements. The

result were communication architectures which were broader in scope, and covered both network and end-system domains. The complete view of QoS provision that covers the QoS management functions on an end-to-end basis on all layers of the distributed multimedia system is called QoS architecture. The rest of this paper will focus on the evolution of QoS architectures from single layer QoS provision, into tentative standards and then full-blown QoS architectures.

2.2 QoS Standards

Earlier QoS work in distributed multimedia literature has been focused on QoS aspects in individual layers of the QoS architecture. A more complete look to the QoS story has started only recently. Since then, several QoS architectures have emerged in the research arena aiming at satisfying QoS requirements for distributed multimedia using a general approach. Due to the short age of the field of QoS management architectures, there are no agreed upon standards for the functionality of a QoS management system. A few standardization efforts have emerged aiming at providing guidelines for building QoS frameworks. This section presents three standards developed so far: ISO OSI, CCITT/ITU and IEEE frameworks.

2.2.1 ISO

The ISO standardization attempts aim at providing QoS support for the widely accepted ISO Reference Model for Open Systems Interconnection (ISO/OSI-RM) communication protocol. The ISO/OSI-RM protocol is a seven-layer protocol with every layer responsible for a phase of communication. Communication occurs by the processing and forwarding of data at each layer to an adjacent layer. The ISO QoS framework [ISO 95] defines a set of QoS characteristics. Applications have the

facility to specify, statically at connection establishment time, the QoS values for the QoS characteristics they desire. QoS management is now a problem of mapping the QoS characteristics across layers and the maintaining the desired QoS level through a set of QoS management functions (QMFs). QMFs are a set of QoS mechanisms that can be combined in several ways in order to meet the defined QoS requirements.

The ISO model defines three levels of agreement for QoS management. An application may request best-effort agreement where no QoS requirements have to be maintained. Another option is compulsory agreement, where QoS requirements are specified but not guaranteed. They may be deliberately degraded to allow for other guaranteed applications to perform. Guaranteed agreements are for applications demanding a certain rate that has to be maintained. Guaranteed applications will not start unless the system is certain they will complete with the required QoS.

The framework also outlines a number of QoS categories for applications to fall into. The QoS categories supplied are: secure systems, safety critical systems, time critical systems, highly reliable systems, easy to use systems, low cost systems, flexible systems and testable systems. Different default QoS policies apply for the different QoS categories presented.

The ISO framework is not a complete framework for QoS management. It does not provide critical solutions to the QoS problems but rather give guidelines on how a complete QoS management architecture should behave. Moreover, the ISO framework relies on simple mapping for network level QoS. It assumes that the network provider will always support QoS provision, which is not always the case.

2.2.2 CCITT (ITU)

CCITT work on QoS is merely to recognize the needs for QoS provision in ATM networks [Hutchison 95]. The CCITT standard provides QoS characterization at three different levels. The call control and connection levels are concerned with the establishment and release of calls and the allocation of resources along the path of ATM switch nodes. The cell control level is concerned with the data transfer phase itself.

CCITT provides a set of manageable QoS characteristics similar to those defined by the ISO framework for equivalent functionality. The QoS characteristics are directly mapped to QoS values for the ATM circuits. It also provides for a process of in-call renegotiations as a form of QoS renegotiations phase.

The CCITT work lacks the view of a complete architecture. It does not show how the QoS characteristics are derived from user-level QoS characteristics above the network layer. There is, also, a deficiency in the definition of how QoS levels are monitored and maintained in the ATM network.

2.2.3 IEEE

IEEE provides guidelines for QoS provision for its definition of the interface requirements of realtime distributed systems communication [IEEE 95]. IEEE discusses QoS with respect to the operating system and the communication system. The operating system provides priorities for applications that are dynamically modifiable to indicate which processes need more realtime data. At the communication layer, four kinds of communication are presented for both unicast and multicast communication: acknowledged, unacknowledged, reliable and unreliable transfers. This is a very vague and qualitative understanding of QoS provision that

needs further revision in order to enable acceptable QoS provision for realtime distributed systems.

2.3 Layer-Specific QoS

Most of the early QoS work in research literature focused on allowing QoS aspects at individual layers of the distributed multimedia systems rather than providing a complete solution for the QoS availability problem in terms of a full QoS architecture. One of two sides was generally considered by early QoS systems: the application level and the transport level.

2.3.1 Application Level

The application level includes both the distributed multimedia system level and operating system running at the end-system. Enhancements for distributed multimedia system aimed at providing new general concepts that would allow the applications to specify their QoS requirements and that would allow the system to obey these requirements. Enhancements at the operating system level aimed at adapting the operating system to provide for the transfer of continuous media instead of the static media support that already exists.

Early experiments aimed at providing QoS support for the ANSA architecture [Campbell 93]. ANSA RPC interface descriptions were modified to include QoS parameters in order for the system to know the applications' QoS requirements. Similar research has been performed at CNET and BBN and Rome Labs [Campbell 96]. Other work aimed at inserting QoS filters [Yeadon 96] at different locations in the distributed system layer to filter the continuous media information to match the

current QoS levels of the system. This is based on knowledgeable filters that understand the continuous media contents and can selectively drop the details out of the media at times of heavy system loading. More recently, the problems for network heterogeneity have been addressed [Banerjea 97] [Gecsei 97]. This aimed at providing QoS with continuous media spanning heterogeneous networks as opposed to the homogenous or single networks addressed in early QoS discussions.

At the operating system level, work has been going on to provide UNIX enhancements or UNIX-like adaptations of operating systems to provide for continuous media transfer and facilitate QoS handling. Significant work has been carried to support continuous media in Amoeba-based UNIX environments, Mach, Chorus, Pegasus and YARTOS [Campbell 96]. The work aimed at optimizing communication protocols and operating system scheduling. Work has also been done to adapt the UNIX SVR4 scheduler to deal with continuous media applications and to provide user level threads in the ARTS operating system [Blair 93]. Yau [Yau 96a] [Yau 96b] has had recent work on I/O efficient buffers, fast system calls, kernel threads and fast direct media streaming to allow continuous media to be effectively handled in the operating system with QoS guarantees.

At the application level, also, comes the matter of storage servers. The main aim was to support simultaneous access to stored services with QoS guarantees. Work has been done on assignment of media to discs, realtime request handling as well as disc layout strategies [Blair 93].

2.3.2 Transport Level

At and below the transport layer, the concern has been to provide the capability of specifying QoS parameters with communications connections and to study network topologies that would allow QoS parameters to be specified and honored.

Several attempts to design transport protocols have been carried out, most of which were later adapted into a full QoS architecture. The HeiTS transport service [Volg 96] was designed to allow for transport QoS and resource management and was later bundled into the HeiProject. The Multimedia Enhanced Transport Protocol (METS) [Campbell 94] [Campbell 97] was designed and later incorporated in the QoS-A. The MMTS protocol [Vogel 95] was later attached to Montreal's architecture. Lately, Yau developed a migrating sockets protocol [Yau 97] as an extension to Berkeley sockets at the user level to provide QoS handling.

Several efforts to provide QoS support to Internet protocols were carried out. RTP [Schulzrinne 95] is a protocol that aims at providing realtime extensions to IP. RSVP [Braden 96] is a resource reservation protocol that runs on top of IP and allows for advanced resource reservation as an extension to IP. IPv6 [Deering 95] is the next generation Internet Protocols that allow for multicast communication and QoS specification for continuous media. IPng allows for extended addressing, QoS handling and enhanced security [Braun 97]. ST-II+ [Delgrossi 95] is a connection oriented network protocol, which allows resource reservation started by the origin of the flow. The associated SCMP protocol allows the initiation and modification of connections allowing resource specifications to be changed dynamically.

Recent research has been performed on network-level topologies to compare and evaluate existing networks with regards to their support for QoS [Stuttgen 97] [Worsley 97].

2.4 QoS Architectures

Earlier QoS work in distributed multimedia literature has been focused on QoS aspects in individual layers of the QoS architecture. A more complete look to the QoS issue has started only recently. Since then, several QoS architectures have emerged in the research arena aiming at satisfying QoS requirements for distributed multimedia using a more general approach. Due to the short age of the field of QoS management architectures, there are no agreed upon standards for the functionality of a QoS management system. A few standardization efforts exist and will be discussed in the early part of this section. The QoS architectures that emerged in research literature will be discussed in the remaining parts of the section. The QoS architectures presented do not all speak the same QoS language. Most presented architectures address QoS management from different viewpoints that are sometimes incomplete when looked at from the perspectives of what has been provided by other architectures. The viewpoints of architectures will be discussed along with the discussion of the architectures themselves.

The current work surveys the following QoS architectures that are available in research literature:

- Tenet Architecture (Berkeley University) [Ferrari 96]
- HeiProject (IBM's European Networking Center, Heidelberg) [Volg 96]

- QoS-A (University of Lancaster, UK) [Campbell 94]
- OMEGA (University of Pennsylvania) [Nahrstedt 95]
- XRM (COMET Group, Columbia University) [Lazar 94]
- Int-serv (Internet Engineering Task Force, IETF) [Braden 94]
- TINA QoS Framework [Bosco 96]
- MASI End-to-End Architecture (Université Pierre et Marie Curie) [Besse 94]
- QoS Framework (Washington University) [Gopalakrishna 94]
- CORBA (Open Management Group, OMG) [OMG 96]
- Siqueria's Thesis (Trinity College, Dublin) [Siqueria 97]
- Univ. of Montreal (Université de Montreal, Canada) [Hafid 96a]
- DJINN (Queen Mary, London) [Mitchell 97]

2.4.1 Tenet Group

The Tenet architecture is an early attempt at providing a complete QoS architecture for QoS management. Nevertheless, the architecture is heavily biased at the network and transport layers with little discussion on the end-system support. The architecture provides a set of protocols to be used for communications. The defined protocols run on top of the Realtime Internet Protocol (RTIP) that is concerned with data transfer. The Continuous Media Transport Protocol (CMTP) runs on top of RTIP and is responsible for the sequenced and periodic delivery of continuous media samples. The Realtime Message Transport Protocol (RMTP) is concerned with message based communication between points. Two control protocols are provided for channel administration and data transfer management. The Realtime Channel Administration Protocol (RCAP) provides generic connection establishment and resource reservation

functions. The Realtime Message Control Protocol (RTCMP) manages data transfers and detects error conditions.

The operation of the protocols occurs in two phases. In the first phase, the source node issues a request to establish a realtime communication channel to a sink source. The request message passes along the nodes connecting the source to the sink. Every node will either accept the message and the QoS level defined in the message or reject it if it cannot attain the required QoS. If the message is accepted, resources are allocated to guarantee the specified QoS level and the message is forwarded to the next node. When the message reaches the sink, the second phase starts by forwarding an acceptance packet back to the source with the agreed upon QoS levels. The nodes in the path make fine adjustments to the allocated resources if needed depending on the agreed QoS. In case of rejection, the nodes free the resources allocated on receiving the rejection message, and forward the message to the previous nodes. The result is either a reserved channel with guaranteed QoS or a rejected request indicating the inability to attain the requested QoS levels.

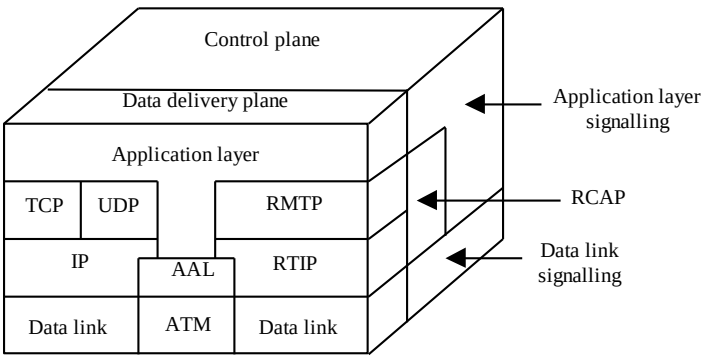


Figure 1 – Tenet Architecture

The defined protocols allow for two kinds of guarantees: deterministic and statistical. Deterministic guarantees provide hard limit bounds on the performance within a session. Statistical guarantees promise a maximum of a certain percentage on delayed and lost packets.

2.4.2 HeiProject

The HeiProject is an advanced QoS model that incorporates both the network level structures with the end-system management structures to provide QoS guarantees.

At the heart of the transport system is a proposed protocol for continuous media transfers, the HeiTS/TP protocol. HeiTS/TP provides the QoS mapping layer and also provides media scaling functions for QoS adaptation. Below the transport layer is the internetworking layer that is based on the ST-II protocol. The ST-II protocol, which was further developed into the ST-II+ protocol [Delgrossi 95], provides both deterministic and statistical service guarantees.

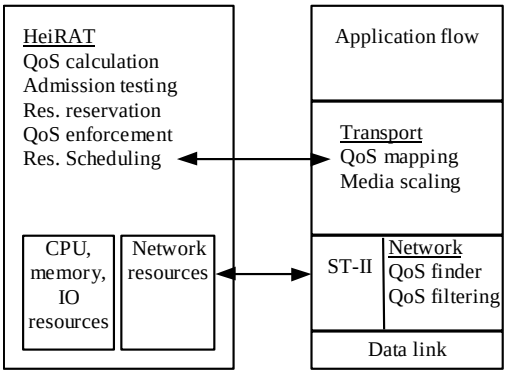


Figure 2 - HeiProject Framework

The end-system support for QoS is accomplished using the HeiRAT management technique. HeiRAT is a resource administration technique that constitutes a full QoS

management scheme that includes QoS negotiation, QoS calculation, admission control, QoS enforcement and resource scheduling. The HeiRAT operating system also takes care of priority thread scheduling.

The HeiProject was designed to handle heterogeneous QoS requests from the individual receivers in a multicast group. This is accomplished through two techniques: filtering and media scaling. In filtering, various network filters are run in different parts of the system, typically at gateways, where they automatically shape the incoming streams based on their knowledge of the stream contents. Media scaling techniques are based on an encoding that allow for progressive representation of data where high quality parts of the data can be dropped at the network areas where low QoS values are required.

2.4.3 QoS-A

The QoS-A [Campbell 93] is a layered architecture of services and mechanisms for QoS management of continuous media flows in multi-service networks. The basic element in the QoS-A is the flow concept, which designates the production, consumption and transfer of media. Flows are always simplex but are either unicast or multicast transfers. They may contain both media and control data.

QoS-A is divided into five layers with QoS functions performed at each layer. The five layers are the network, physical, data-link, transport, and distributed platform layers. The distributed platform layer is the highest layer and contains a multimedia interface that allows for QoS configuration, multimedia functions, synchronization mechanisms and realtime scheduling mechanisms. The orchestration layer provides services that control temporal parameters of flows such as jitter and transfer rate. It

also provides primitives that ensure synchronization. QoS-A also defines a special protocol for the transport layer to allow for QoS provision, the Multimedia Enhanced Transport Service (METS). METS provides an interface for QoS to be specified, negotiated and contracted. METS allows for deterministic, statistical and best-effort communication.

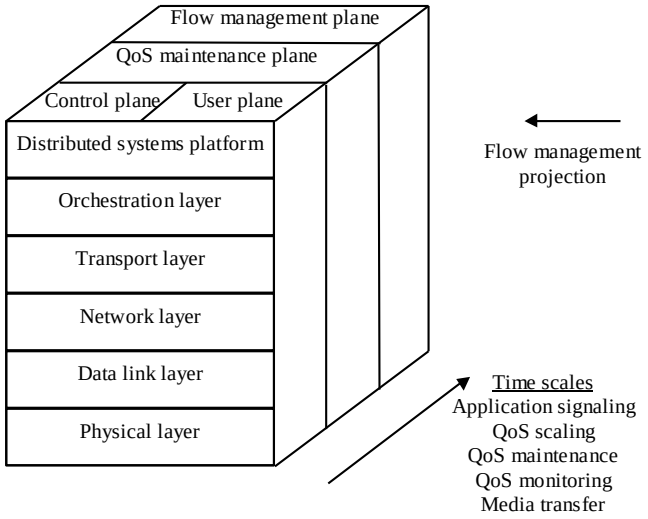


Figure 3 - QoS-A Framework

Each of the five layers is represented in three different planes: protocol, QoS maintenance and flow management planes. The protocol plane consists of a user plane and a control plane. Both planes are separated due to the different QoS requirements for both user data and control data. The QoS maintenance plane contains layer-specific QoS managers for monitoring and maintaining the associated protocol entities. QoS managers are responsible for the maintenance of the QoS contract. The flow management plane is responsible for the flow establishment, QoS renegotiations, QoS mapping and QoS adaptation.

2.4.4 OMEGA

The OMEGA architecture [Nahrstedt 95] aims at including the application layer with the end systems and network to the QoS management mechanism. By including the application layer, OMEGA aims at providing a simple QoS specification phase for its applications, relieving the applications of the burden of having to deal with complicated QoS parameters and values. End-system negotiation is a part of the internal working of the OMEGA system. It is performed during session establishment between the applications. QoS parameters negotiated in OMEGA are parameters for the application as a whole. This is to be contrasted with other architectures that aim at providing per-flow QoS management.

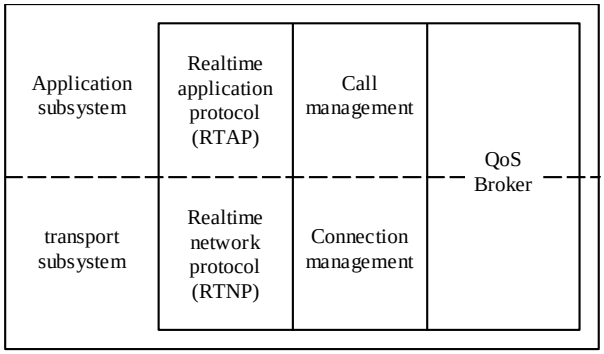


Figure 4 - OMEGA Framework

OMEGA assumes a network that is capable of delivering packets with a bound on the delays and errors as well as a guaranteed bandwidth. This way, OMEGA frees itself from the hassles of having to simulate guaranteed throughput on a non-guaranteed network. OMEGA provides an application layer and a transport layer. The application layer provides functions for connection establishment and rate management. The transport layer provides basic functions, such as connection establishment and termination. OMEGA keeps a QoS broker that negotiates the guarantees required by

the applications with the guarantees provided by the network and transport layer. The QoS broker has a QoS buyer and QoS seller. The buyer receives high-level requests from applications and sends resource queries for the QoS buyer to reserve resources. The seller manages resources and answers the queries sent by the buyer. The QoS broker handles the admission control functions internally.

2.4.5 XRM

XRM is an architecture that provides QoS management at end-system and network levels through dividing its architecture into five planes.

- The network management plane is responsible for OSI model of communication.
- The resource control plane provides cell scheduling and call-management in the network. It also handles memory management and admission control at the end-system level.
- The connection management and control plane handles running connections and traffic control functions.
- The user transport plane is responsible for providing a multimedia transport interface for end-systems that is capable of transferring multimedia information.
- The data abstraction and management plane represents abstractions of the data provided in the system. It implements data sharing among all other planes.

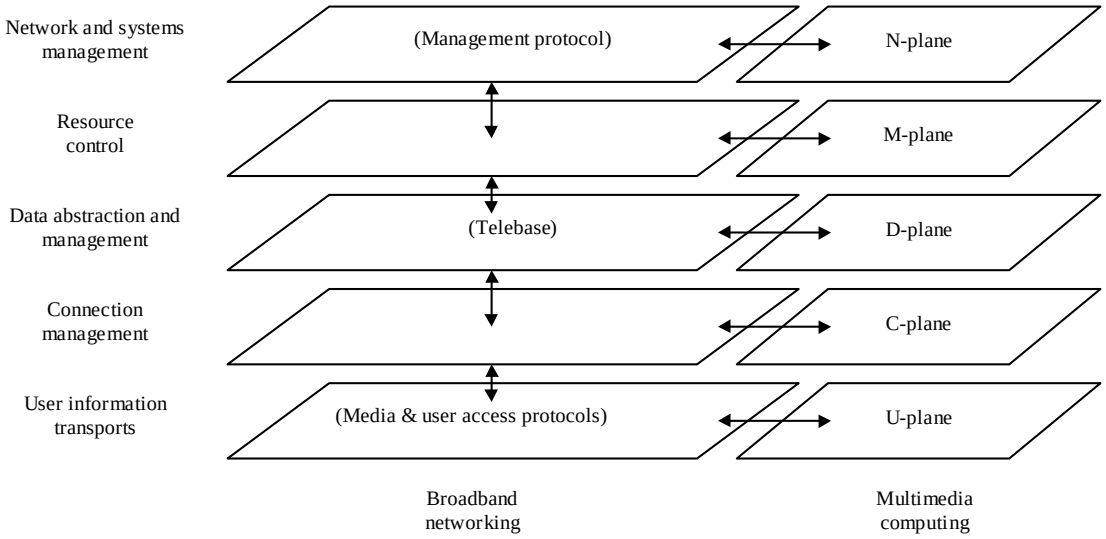


Figure 5 - XRM Framework

XRM is based on a network that provides QoS guarantees such as ATM networks. Packets in XRM are classified into one of four classes (class I, class II, class III and class C). Each class represents a set of QoS requirements. QoS requirements are met through providing algorithms for cell scheduling and buffer management to dynamically manage communication of cells.

At the end-system level, applications specify QoS requirements in terms of known standards such as MPEG-I video or CD-quality audio. The QoS requirements for the known classes are translated into QoS parameters to the system to provide QoS guarantees for the application.

2.4.6 Int-serv (IETF)

The Internet Engineering Task Force proposed a QoS management architecture that attempts to provide QoS management to Internet communication. The framework tries to add QoS management to the various elements involved in communication making

them “QoS aware.” The architecture provides models for QoS management in various network elements, such as routers and sub-networks and end-systems. The architecture is essentially a network level architecture that could be adapted to end-systems.

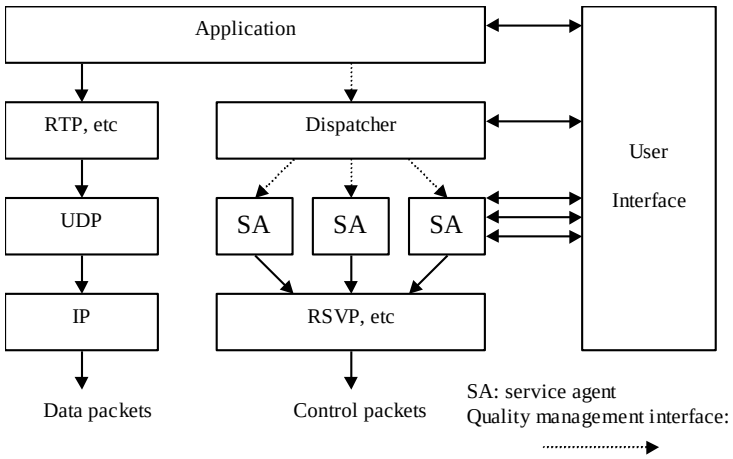


Figure 6 - Int-serv Framework

IETF provides for four types of delay: best effort, controlled, predicted and guaranteed. Best effort delays are the normal Internet delays that depend on the current system and network load levels. Controlled delay provides for a choice of one of several typical delays. Predicted delay provides a statistical delay bound. The guaranteed delay is an absolute guaranteed value for delay.

At the application level, applications submit two sets of information to the QoS manager. First, the application provides the traffic specification, which indicates the patterns of use the application expects. Later, the application provides the service request specification, which shows the QoS requirements for the application from the different elements of the system.

The IETF architecture is made up of four components:

- The packet scheduler handles packet communication based on queues and timers.
- The classifier is responsible for grouping the packets into their respective QoS levels.
- The admission controller is responsible for QoS parameter calculations and connection acceptance or refusal.
- The reservation setup protocol is responsible for reserving resources along the path of the flow during the existence of the connection.

The architecture later describes a QoS manager (QM) which acts as an abstract layer that separates applications from the details of underlying networks enabling applications to specify QoS requirements independently.

2.4.7 TINA

The TINA QoS framework is based on the differentiation between telecommunications applications and the distributed processing environment. Applications specify their QoS requirements in terms of service attributes in the context of the Computing Architecture [Aurrecoechea 98]. Resource managers employ QoS mechanisms to adhere to agreed QoS contexts. By separating applications from the processing environment, QoS declaration can be performed without having to deal with the complexes of resource management mechanisms required by the system.

The TINA project was further enhanced [Bosco 96] to support realtime and multimedia traffic through the use of the CORBA and ODP (Open Distributed Processing) standards, while keeping compatibility with the original TINA architecture. The new project is called ReTINA and is funded by Chorus Systems, Alcatel, Siemens, HP, CSELT, France Telecom, British Telecom, Telenor, APM, O2 Technology and Broadcom. The project provides extensions to the CORBA IDL to incorporate stream abstractions.

2.4.8 MASI

The MASI project [Besse 94] aims at developing end-to-end QoS support in multimedia systems. The QoS framework specifies QoS requirements at the application level and considers resource management at the application level as well as at the transport and network levels. MASI operates on ATM-based networks.

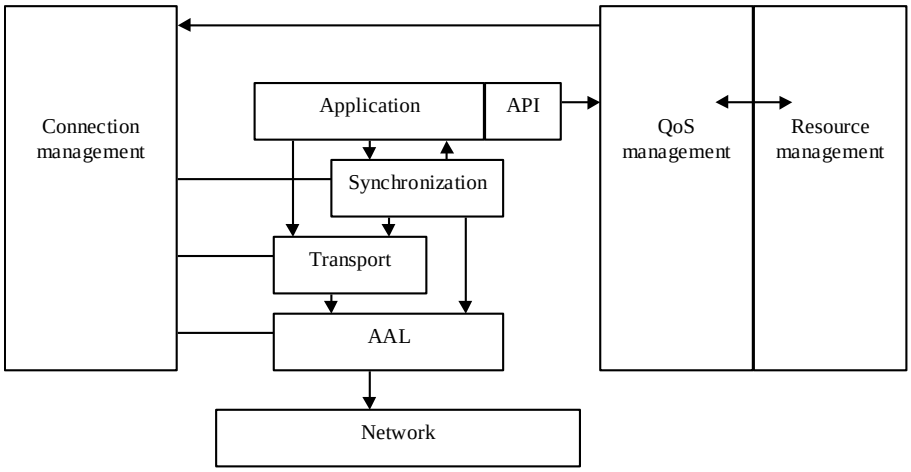


Figure 7 - MASI Framework

MASI research is motivated by the following parameters [Aurrecoechea 98]:

- The need to map QoS requirements from the ODP layer to specific resource modules efficiently and cleanly.
- The need to resolve multimedia synchronization issues.
- The need to provide suitable support for the communication protocols for multimedia services.

2.4.9 Washington Univ.

The Washington University model is based on QoS specification for end-to-end systems. The system provides four main functions: QoS specification, QoS mapping, QoS enforcement and transport-level realtime communication. QoS specification is performed through providing a limited set of QoS parameters for the applications to specify to make it simpler for applications to state their requirements. QoS mapping translates the QoS values into network level resources to be managed. Three types of resources are managed in this architecture: CPU, memory and network. QoS enforcement aims at providing realtime performance guarantees for the applications. This is performed using realtime upcalls (RTUs). RTUs are a means of transferring control to the system through a rate monotonic policy. This eliminates the need for frequent context switching and provides the rate required for realtime communication. RTUs are also used in sending packets at the transport layer at rates implied by the QoS values specified by the applications.

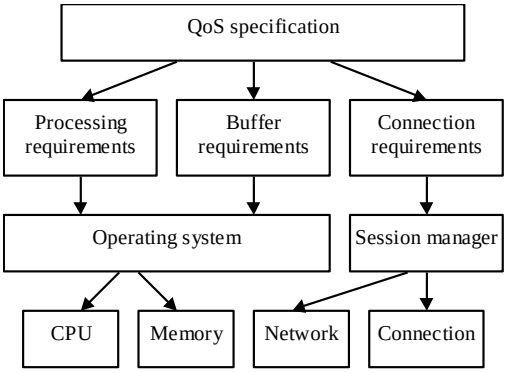


Figure 8 - Washington U. Framework

2.4.10 CORBA

The Object Management Group has created a Special Interest Group on QoS management in CORBA. The QoS SIG proposed a QoS architecture [OMG 96] that extends CORBA to support QoS management. The proposed architecture allows objects to specify their functional and non-functional requirements as multi-valued parameters. This allows clients to request more from objects than the normal binary (yes/no) behavior. Clients request data from objects based on QoS levels that the object can supply. The system allows the client to request the required QoS along with the service request. The QoS architecture is responsible for maintaining the agreed QoS levels within specified system constraints.

The QoS manager in CORBA has the following goals:

- Allowing clients to request and renegotiate QoS.
- Allowing servers to describe QoS characteristics and QoS ranges that they can provide.

- Arbitrating resources in order to contain the system's acceptable behavior within the desired bounds and to enforce QoS guarantees.

The CORBA QoS manager proposes the concept of performance polymorphism as a technique to provide several QoS levels. The CORBA objects have their methods overloaded to provide different outputs at different operating QoS levels.

2.4.11 Siqueria's Thesis

Frank Siqueria [Siqueria 97] proposed the adoption of CORBA on top of IP networks to provide for QoS management. The proposed architecture allows the transmission of control data using CORBA as a middleware on TCP/IP networks and stream data using more advanced IP protocols. Siqueria proposed the use of IPng along with RTP and RSVP on Integrated Services Networks. RTP (Realtime Transport Protocol) and RSVP (Resource ReSereVation Protocol) are newly produced Internet protocols that do not have middleware support yet. The architecture provides a transparent layer that the applications use to handle stream data.

Siqueria's improvements on the adopted core are:

- The implementation of the CORBA stream mechanism on novel Internet protocols.
- The definition of sets of QoS parameters for different categories of multimedia applications.
- The definition of algebra for translating application QoS parameters to network-level QoS parameters or resource reservation messages whenever possible.

- The provision of a multimedia component hierarchy addition to the CORBA framework to facilitate the construction of distributed multimedia applications using CORBA.

2.4.12 Univ. of Montreal

Hafid and Kerherve [Hafid 96a] proposed a QoS architecture that provides application-level and transport-level QoS management. The architecture is composed of three QoS interfaces for the client, server and transport systems. In addition to the QoS interfaces, the architecture employs a transport protocol that allows for the QoS negotiation of the three parties. The client interface is a qualitative interface that allows the application to set its QoS requirements in terms of belonging to specific QoS groups, such as video, audio or still images. Server QoS interfaces allow the server to specify similar information for the multimedia documents it is capable of providing. The architecture provides its own multimedia transport service (MMTS) that allows for connection oriented services and unidirectional point-to-point transmission with adhering to given QoS parameters. QoS parameters at the transport level are quantitative values that map directly to the network level. The architecture provides an extensive QoS negotiation mechanism [Kerherve 94] that works with MMTS.

2.4.13 DJINN

The DJINN framework [Mitchell 97] developed at Queen Mary tackles the QoS problem for groupware applications. Groupware have different requirements than those of normal static distributed multimedia applications. Groupware rely on the dynamic functions of several cooperating applications with applications joining and

leaving communication channels dynamically. This has implications on admission control and resource reservation policies.

The DJINN system takes the approach of developing a QoS model for the application where high-level QoS requirements are captured and expressed. The encapsulated QoS properties form a natural form for later reconfiguration. The QoS model is separate from the application. This allows the model to be created and admission control performed on the model before the actual application is created. This saves the trouble of creating the application, which could be a long and remote process.

Chapter 3 TRANSPORT-LAYER PROTOCOLS

3.1 Introduction

The QoS architectures introduced in the previous chapter incorporate a transport level protocol to provide the heart of the guaranteed communication service in distributed multimedia systems. Transport protocols are the fourth layer of the seven-layer ISO OSI model shown in Figure 9. They provide the basic end-to-end communication requirements of collaborating entities and applications. Other layer protocols, although needed, are less important to designers and far less complex [Stallings94]. Applications can easily be programmed to access the transport layer directly to achieve its communication requirements. This is the normal mode of operation for the DOD's transport protocols as well as all the QoS architectures surveyed in the previous chapter.

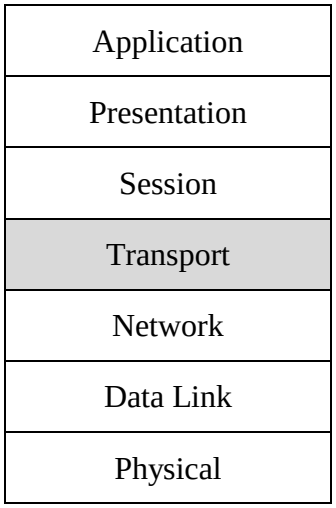


Figure 9 - ISO OSI Model Layers

In the past two decades, there were huge leaps in network architectures and important advances in physical network layers. Unfortunately, the physical layer advances were not matched with an equivalent progression in the design of the transport protocols that operate on the new physical layers. This reduced the benefit of the great physical advances to the application users, as the introduction of highways does not help a lot if bicycles are still used for transportation. New transport protocols had to be individually developed by QoS architectures to provide their end-to-end communication means. Existing protocols could not be used due to their lack of QoS definition, which now exists in many physical layers such as FDDI and ATM. The new protocols make use of the QoS-enabled hardware to provide QoS-enabled protocols.

QoS-enabled transport protocols are a requirement for distributed multimedia communication since the advancement in hardware alone does not eliminate the need for a matching advancement in transport protocols. Braden [Braden 94] answers the myths raised that QoS-enabled transport protocols are not required because the bandwidth will eventually be infinite causing the existing protocols to suffice. It is impractical to assume in the short or medium term that bandwidth will be so abundant and cheap that there will be no communication delays other than the speed of light eliminating the need to reserve resources. While raw bandwidth may seem inexpensive, bandwidth provided as network service is not likely to become so cheap that wasting it will be the most cost-effective design principle. Unless we provide for the possibility of dealing with congested links, then realtime applications will simply be precluded in those cases. Furthermore, simple priority provided by existing protocols is insufficient. Existing priority levels are not likely to be adequate for

defining all realtime streams in the future. If too many applications are tied together in a single priority level, they will all compete for the priorities of that level causing degradation for all applications in the level. Unless some quantitative means are provided to differentiate among applications, no true prioritization can be achieved. This adds to our belief that enabling QoS functionality in transport protocols is necessary for meeting the requirements of advanced multimedia applications of the current and upcoming decades.

3.2 Transport Services

Transport protocols operate by providing a set of services that are available to higher level layers and user applications. The specified services shield the user from the underlying details of the lower layers. The transport protocol, in turn, makes use of the services supplied by the available lower layers to provide its named services. The transport protocol provides the named services through a set of known access points; each called a Transport Service Access Point (TSAP). The users of the transport service utilize a TSAP to communicate with the transport entity as shown in Figure 10. The transport entity abstracts the details of lower layers to provide the same functionality to all its users irrespective of the underlying layers.

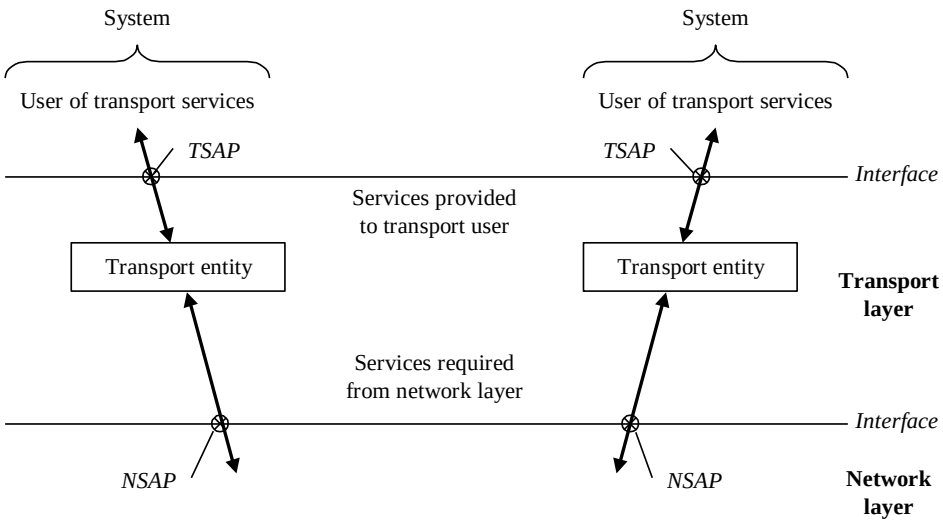
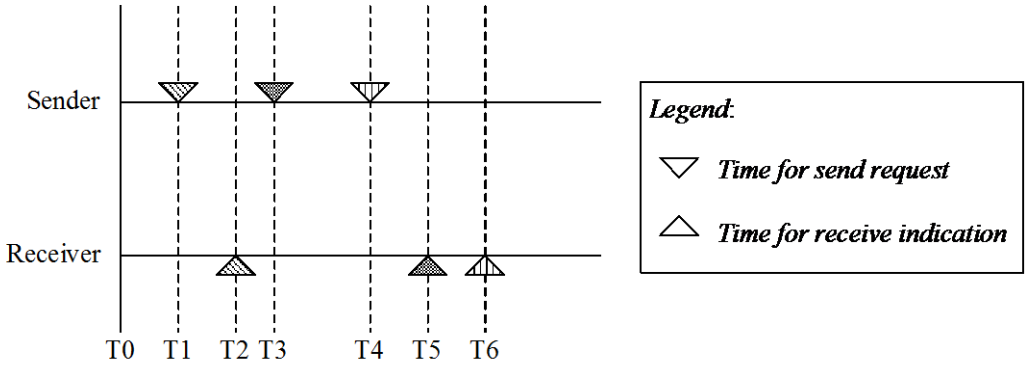


Figure 10 - Transport Service Dynamics

Modern transport protocols are expected to provide QoS guarantees for the intelligible transmission of continuous media in distributed multimedia systems. Current transport protocols, which do not provide QoS support, will not be able to correctly manage coexisting flows of traditional burst data and continuous media. At many times, traditional burst data will overwhelm the transport service with large amounts of data that is not time-critical. The transport protocol, not being able to differentiate, will send the data that comes to it first, delaying the other until the send is over with no clever multiplexing of data on the shared communication link. The cuts in continuous media transmission in order to send the traditional burst data will cause the continuous media to arrive after a delay. This delay is unacceptable for continuous media since it breaks the continuity of multimedia playback. QoS guarantees help the application in obtaining the desired level of service to ensure correct playback. They also help the transport protocol to correctly multiplex data on the shared communication link to

satisfy the needs of different kinds of applications as permissible by the available resources.

Communication in distributed multimedia systems is allowed in either the single sender to single receiver mode, known as *unicast*, or in single sender to multiple receiver mode, known as *multicast*. A single binding between a sender and one or many receivers is termed a *continuous media data flow*, or shortly a *flow*. Transport protocols primarily require the definition of several quantitative QoS parameters to characterize continuous media flows in order to be able to manage them correctly. The most important parameters to consider are throughput, transit delay, transit delay jitter and error rate. These parameters help the transport protocol in scheduling the transmission of the flows' data units, named Transport Service Data Units (TSDU). Figure 11 provides a graphical rationalization of the main QoS parameters.



Throughput for TSDU#1 = min [L1 / (T3-T1), L1 / (T5-T2)]
Throughput for TSDU#2 = min [L2 / (T4-T3), L2 / (T6-T5)]

Transit delay for TSDU#1 = T2-T1
Transit delay for TSDU#2 = T5-T3
Transit delay for TSDU#3 = T6-T4

Transit delay jitter = (T5-T3) – (T2-T1)
(assuming the time line is drawn to scale)

Figure 11 - QoS Parameters Definition

- Throughput is a quality that defines the rate at which TSDUs are transmitted from the sender to the receiver. It is defined in the ISO standard [ISO8072] as the smaller of the sender's throughput and the receiver's throughput. The sender's throughput is the TSDU size divided by the time interval between the previous and the last TSDUs being presented by the sender. The receiver's throughput is the TSDU size divided by the time interval between the previous and the last TSDUs being indicated for receipt by the receiver.

- Transit delay refers to the time between the signaling of a TSDU-send invocation on the sender side and the receipt of a TSDU-receive indication on the receiver side.

- Transit delay jitter, or shortly jitter, is the variance in transit delay. Jitter is defined as the difference between the longest and the shortest transit delays observed for TSDU transmissions for a flow.

- Error rate is a measure of the tolerance of the flow to communication impairments. Continuous media flows tend to have stringent timing parameters with more relaxed error rates. This allows for TSDUs to reach the receiver in a timely fashion but not necessarily in a reliable way. TSDUs arriving with errors simply appear as noise during the playback of the media stream. Burst data, on the other hand, tend to have relaxed timing requirements with strict error rate demands.

The commitment of the transport protocol to the contracted QoS values for a flow is not necessarily a guarantee of full abidance. The strictness of abiding by the contracted QoS context depends on the QoS policies employed by the transport

protocol. The contracted QoS values can be totally ignored to provide *best-effort* QoS. This is the case with the current transport protocols. QoS values can also be used as *predictive* tools to aid the transport protocol in scheduling its TSDUs. The most common use for QoS values is to provide *statistical* guarantees on TSDU delivery, where it is guaranteed that a certain contracted percentage of the TSDUs will exhibit the contracted QoS values. The most stringent manner for applying the QoS values is to provide full *guaranteed* QoS delivery. This ensures that all TSDUs in a flow will be delivered with the contracted QoS values. Although, at first sight, this might seem the only logical way to implement QoS provision, guaranteed QoS delivery has severe impact on resource utilization patterns. Resources have to be reserved for the single usage of the contracting flow with no sharing even if they will be wasted most of the time.

The transport layer provides QoS management at different stages during the lifetime of a flow. The first QoS role played by the transport protocol is during connection establishment. At connection establishment, the service user files a request for flow setup at a TSAP using a set of QoS values, called QoS context. The transport protocol validates the context and performs admission control strategies to either accept or reject the establishment request. If the resources required for the requested QoS context are not available, the service user is notified to start a series of QoS negotiation ending in either a modified QoS context that can be accepted or the cancellation of the establishment request. Subsequently, resources are reserved along the path from the sender to the receiver to ensure the contracted QoS context according to the QoS policies of the transport service. The managed resources are the network bandwidth, protocol buffers, and CPU cycles. During the lifetime of the flow,

TSDUs are accepted from the sender, classified by the transport service in one of it sending queues and routed appropriately to reach its receiver. TSDUs are scheduled according to the timing constraints of the flow's QoS context. TSDUs may be dropped, within the allowable error rate, to meet the QoS constraints of the current flow and other flows in the network. The transport protocol employs flow control mechanisms on the TSDUs transmitted from the sender to the receiver in order to ensure proper usage of the reserved resources. QoS monitoring is performed along the lifetime of flows to indicate any failures in achieving contracted QoS contexts. Upon failure detection, QoS degradation signaling can be performed to indicate QoS problems to the contracting user. The result can vary from ignoring the signal, starting QoS re-negotiation, or flow termination. At any time, the flow creator may ask for flow termination. The transport layer processes termination requests and releases all reserved resources related to the flow being terminated.

Transport protocols are expected to provide QoS irrespective of the underlying network. The existence of a network layer that supports QoS relieves the transport layer from many burdens. Nevertheless, network layers that support QoS are not widely spread nowadays and they are not expected to be dominant in the near future. The vast majority of existing LANs still rely on Ethernet technologies, which are not QoS aware. This adds to the complexity of the QoS-aware transport layer. QoS-aware transport protocols should be able to deal with the heterogeneity of the existing Internet.

The rest of this chapter surveys a number of transport protocols being developed in current research literature. Some of the protocols discussed were developed mainly for

QoS architectures while others are enhanced protocols for the Internet. The protocols developed for QoS architectures assume an QoS-aware network layer in order to provide QoS guarantees. The presented Internet protocols provide only guidelines for realtime scheduling of TSDUs and predictive QoS management. The chapter concludes with a description of what is needed for a transport protocol that is capable of providing QoS-aware functionality irrespective of the underlying network layers.

3.3 Transport Protocols in QoS Architectures

The QoS architectures presented in the previous chapter rely on an integrated transport protocol that provides the core functionality of QoS provision. The developed transport protocols vary in several aspects. Not all protocols provide guaranteed delivery policies. Furthermore, several protocols provide their own flavor and combination of QoS policies to create new policies. The mechanisms employed by the transport protocols are diverse. Connection establishment and termination mechanisms vary from handshaking to timer-based connections. The employed flow and error control mechanisms are another point of concern. The presented protocols also differ in the type of QoS monitoring and signaling employed as well as the QoS negotiation models available.

3.3.1 OSI95

The OSI 95 Project [DBL94] was developed at the Université de Liège in Belgium. The project provides a transport service with multimedia support. QoS enhancements in OSI 95 include a new set of negotiation mechanisms as well as a set of QoS policies for the transport protocol to provide. The transport protocol allows for QoS specification, negotiation and monitoring. QoS re-negotiation facilities are not

provided. OSI 95 does not require an QoS-aware network layer and, in turn, it does not guarantee bounds on service. However, the obligation is on the behavior when the service bounds are not satisfied.

OSI 95 provides three types of QoS policies: compulsory, threshold, and maximum quality. Compulsory QoS values are specified when the required values are to be strictly adhered to. The transport service terminates a flow if it cannot maintain the contracted compulsory QoS values at the required level. Threshold QoS is similar to the best-effort QoS provided by traditional transport protocols except that the transport protocol is obliged to signal the service users when the instantaneous QoS of the flow falls below the specified threshold QoS. Maximum quality QoS is provided to limit the resources used by the transport layer for a particular flow. Normally, flows will not complain if the actual QoS is more than what is required. Maximum quality QoS is based on the philosophy of minimizing unnecessary usage for cost-saving purposes. This suits environments where users are charged for the resources used.

OSI 95 provides two negotiation mechanisms to specify the three QoS values of a flow. Both negotiation mechanisms are based on the classical 4-primitive exchange: request, indication, response, and confirmation as shown in Figure 12.

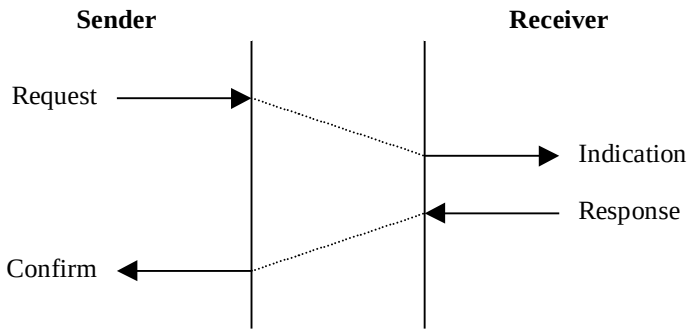


Figure 12 - The Classical 4-Primitive Exchange

The first mechanism is the "Triangular Negotiation for Information Exchange". In this type of negotiation, the sender suggests a QoS value in the request primitive. The transport service may weaken the suggested value before passing it to the receiver in an indication primitive. The receiver may further weaken the QoS value in the response primitive. The sender finally accepts or rejects the final QoS value in the confirm primitive. This type of negotiation is performed for the maximum quality QoS values.

The second mechanism is the "Triangular Negotiation for a Contractual Value". The goal in this type of negotiation is to find a contractual value that binds both the service provider and its users. The user provides a minimal requested QoS value and a bound for strengthening this value. The service provider may decrease this bound in the indication primitive. The receiver may finally reduce this bound to select a QoS value that is still stronger than the minimal value. The sender finally accepts or rejects the selected QoS value in the confirm primitive. This negotiation mechanism is used for negotiating compulsory and threshold QoS values.

3.3.2 QoS-A

The QoS-A [Campbell94a] developed at Lancaster uses a locally developed Multimedia Enhanced Transport Service (METS) [Campbell94b] to provide its transport layer requirements. METS provides access points for end-to-end multicast communication of service users. QoS levels are contracted through well-defined primitives that build a service QoS contract. Service contracts QoS levels are achieved in the context of a local ATM network.

METS works with the notion of flows. A METS flow is an end-to-end communication binding between a sender and its receivers. Service users establish a continuous media flow with an agreed service contract via the following primitive:

```
FLOW_ID    FLOW_CONNECT_REQUEST

   ( TSAP_T *SOURCE, *SINK ; SERVICE_CONTRACT_T *QOS );
```

The sink TSAP may also represent a group address to accommodate multicast flows. The service contract subsumes the well-accepted QoS parameters of jitter, error, delay and throughput, but also allows the specification of a wider range of options. These are characterized in terms of the following clauses:

- *Flow_spec_t* characterizes the user's traffic performance requirements.
- *Commitment_t* specifies the degree of resource commitment required from the lower layers.
- *Adaptation_t* identifies actions to be taken in the event of violations to the contracted services.

- *Maintenance_t* selects the degree of monitoring and active QoS maintenance required of the QoS-A.

- *Connection_t* selects from negotiated, fast reservation and forward reservation connection services.

- *Cost_t* the costs the user is willing to incur for the services requested.

Flow management is the key function of the METS protocol. METS provides two kinds of TSAPs: flow management (FM-TSAP) and data (DATA-TSAP). Flow management data taken from the FM-TSAP are sent using a separate out-of-bound channel used for control and signaling. Each FM-TSAP provides the primitives of get_tsap, free_tsap, flow_connect, qos_renegotiation, qos_degradation, qos_report, monitor_flow, flow_assessment, maintain_flow and flow_disconnect. Every DATA-TSAP provides the primitives data_request, data_indication, data_response and data_confirm.

METS works in three planes: the protocol plane, the QoS maintenance plane and the flow management plane. The protocol plane consists of a flow regulator, a flow scheduler, a flow monitor and a resource manager. The QoS maintenance plane is responsible for maintaining and monitoring QoS values for periodic QoS assessment. The flow management plane is composed of a signaling infrastructure, a flow reservation protocol and a QoS adaptation layer.

3.4 Internet Transport Protocols

The Internet is based on the historic Internet Protocol (IP) developed more than two decades ago. The fundamental goal of IP was to provide an efficient protocol capable

of interconnecting heterogeneous networks without any change. It was based on datagrams, which held all the state required for their routing and transmission. Routers and gateways along communications paths were stateless and did not need to carry any information about the established communication flows. This was critical to provide the fate-sharing model of IP. The fate-sharing model is a philosophy where the information of an end-user may be lost only if that end-user is lost, too. It also satisfied the goal of robustness where intermediate nodes were allowed to break down without affecting existing communication paths. Two protocols were developed on top of IP: TCP and UDP. UDP provides a connectionless datagram service that allows sending small packets of data in an ordered fashion. Routers can drop datagrams in case of network congestion. TCP provides a reliable, connection-oriented service that allows a path to be established from sender to receiver. Packets flow along the established path in a reliable and ordered fashion. IP, in its first specifications, does not provide for multicast communication. IP also lacks QoS provision.

Two major attempts to add multicast and QoS support to IP are ST-II and RSVP. ST-II and RSVP are discussed in this section.

3.4.1 ST-II

STream protocol II (ST-II) [Delgrossi95] models a resource reservation as a simplex data stream rooted at the source and extending to all receivers via a multicast distribution tree. ST-II provides for multicast group creation and does not assume that the underlying layer supports multicasting. Streams are initiated by Connect-requests at the sender. Connect-requests travel along the paths of a network until they reach the receiver. Along the communication path, routers reserve the appropriate resources

specified by the Connect-request and update the request with lower values in case the original values cannot be met. The receiver finally accepts or rejects the values that arrive in the Connect request and propagate the packet back to the sender in order to confirm the resource reservation. A multicast channel is built along the communication path in a tree structure. Receivers refuse Connect requests by sending a Disconnect message. Group communication is allowed by adding and removing receivers dynamically after the initial stream setup.

ST-II routers maintain hard state for the active connections in a network by creating a virtual circuit. This strategy defies the original philosophy of fate sharing in IP. Control messages in ST-II are sent using reliable, acknowledged transmission. A Hello protocol is employed to periodically check the reliability of an already existing connection from host to host. When a change in host routing is detected due to a failure in an intermediate node, stream recovery procedures are attempted. ST-II does not provide means for QoS monitoring and QoS signaling in case of degradation. QoS values specified in Connect requests are only used as a predictive measure for resource reservation.

3.4.2 RSVP

The Resource reSerVation Protocol (RSVP) [Braden94] is another enhancement to the IP protocol. It also employs the concept of a path along which resources are reserved to provide better QoS-enabled communication. RSVP is different from ST-II in that it does not provide its own multicast mechanisms. RSVP relies on the multicast capabilities of the underlying layers.

RSVP flows are receiver-initiated. Receivers start a Path request to join a multicast flow group and identify its flow requirements. This allows different receivers to define different QoS context for the same flow according to its capabilities. This way, receivers with less-capable networks can ask for lower QoS levels in the same multicast flow.

RSVP incorporates a datagram messaging protocol with periodic refreshes to maintain soft state in the intermediate switches to provide reliability and robustness. Soft state, as opposed to hard state in ST-II, means that path data are stored in intermediate switches temporarily. Soft state is deleted if not periodically refreshed. This allows orphaned reservations to be deleted automatically. Maintaining soft state, as opposed to hard state, complies with the fate-sharing model of IP. Soft state is also more robust because it allows intermediate nodes to fail without interrupting already existing flows.

RSVP models a reservation as two distinct components: a resource allocation and a packet filter. The resource allocation specifies what amount of resources is reserved while the packet filter selects which packets can use the resources. This allows for sharing resources among flows.

Chapter 4 PROPOSED ARCHITECTURE

4.1 Overview

Research in the field of QoS management is mostly directed towards either providing a complete QoS management architecture, or enhancing one or more of the architecture aspects discussed in the previous chapters. Most QoS research assumes a network that is capable of delivering packets with a bound on the delays and errors as well as a guaranteed bandwidth. The research is mainly aimed at utilizing the existing QoS features of modern QoS-aware platforms, such as ATM networks. Little research has been directed towards non-QoS-aware platforms, which, unfortunately, comprise most existing physical networks.

The majority of QoS research for non-QoS-aware networks adopted one of two approaches: resource reservation and/or notification. In the resource reservation approach, mechanisms are developed to allow allocating resources to data flows on platforms that do not inherently provide support for resource allocation and reservation. This allows non-QoS-aware networks to be transformed into being QoS-aware by dividing their resources among existing and expected data flows and hence allowing the mechanisms developed for QoS-aware networks to be used on non-QoS-aware platforms. The other approach was to develop software notification mechanisms for data flows that exceed the capabilities of the underlying network. This allows data flows to cooperatively work towards reducing their requirements at times of peak demand.

The present work aims at bringing QoS awareness to platforms that do not have inherent support for QoS. **The goal of this thesis is to borrow, adapt and develop mechanisms for porting the QoS concepts that were originally developed for QoS-aware platforms to work with the non-QoS-aware platforms.** The approach used in this thesis is a combination of the *complete QoS-architecture model*, originally developed for QoS-aware platforms, and a *modified notification approach*. The approach includes a heuristic admission control mechanism to avoid accepting data flow requests that the underlying network will probably not be able to serve. The benefit of this approach is that it does not make any assumptions about the underlying network, its capacity, or any applications that are currently running on it. This allows controlled QoS flows to coexist with non-controlled data on the network, such as an FTP session.

A complete QoS architecture is designed including the essential aspects of a QoS architecture. The proposed architecture provides for QoS specification, mapping, maintenance, monitoring and notification. The architecture also allows for QoS admission based on fuzzy logic and a neural network. The QoS admission mechanism decides whether new data flows should be accepted or not based on the parameters of the new flow and the existing network status as monitored in the existing flows.

In order to verify the results of the designed architecture, a prototype QoS architecture is developed in C++ under the Linux operating system. The prototype includes an implementation of all the proposed QoS architecture aspects as well as the neuro-fuzzy admission control mechanism. A distributed multimedia application is also developed and deployed across a non-dedicated LAN where other uncontrolled data is

running. The application demonstrates the effectiveness of the designed solution in allowing more multimedia flows to arrive comprehensibly to their destinations than an uncontrolled system would permit.

4.2 Design

The design of the proposed system adopts the concept of a *flow*. **The flow is a single logical instance that binds a sender with a receiver.** The flow is used to pass multimedia information from the sender to the receiver. It is similar to a data connection between two points except for the fact that the data passed is time-sensitive multimedia data.

The proposed system provides a middle layer between applications and the operating system to add QoS awareness to the operating system transport protocols. This middle layer allows for the creation, maintenance and termination of flows.

The system design follows the client/server model. Servers (daemons) that take QoS-related decisions and perform flow-prioritization provide the QoS functionality of the middle layer. The clients of the system (applications) use a system library that provides QoS-related functions to communicate with the QoS servers. The system library is made up of stubs that perform remote procedure call (RPC) connections to the servers to complete the required operations on behalf of the clients.

4.2.1 Goals

The design of the system was made with the following goals in mind:

- **Transparency to the user:** Sending and receiving applications should manipulate QoS-aware multimedia flows in the same manner that normal data is handled.

Multimedia data that was originally transmitted using traditional communication should be easily ported to the concept of flows.

- **No network utilization assumptions:** The proposed QoS system should neither assume that it is the sole entity that utilizes the underlying network, nor assume a certain level of network utilization. The QoS-aware flows should coexist with current non-QoS-aware data being transmitted on the network, such as FTP sessions and burst application communication. No utilization pattern shall be assumed for the underlying network.

- **Avoiding bottlenecks:** The design of the system should avoid having a single entity where QoS-related information is stored. This avoids having a single point of failure to increase the fault-tolerance of the system. This design would also reduce the possibility of a having a bottleneck that would degrade the performance of the system as a whole.

- **Application-level structures:** The proposed system should all run in user space. The system should make use of current traditional transport protocols provided by operating systems by building on them rather than modifying them at the system level. This allows applications that were already developed to use existing traditional protocols to run unmodified when the new additions are used.

4.2.2 Decisions

In order to achieve the design goals defined earlier, the following design decisions were made:

- **Out-of-bound control:** Control information sent between the different system elements will be sent on a channel different from that used by the flows. The flow will only be used to send multimedia information sent by the system users. Any control data is sent directly using the underlying non-QoS-aware transport protocol.

- **Sender-initiated flows:** Flows are to be created by the sender of the multimedia data. The receiver of the data should be expecting the data and waiting for it.

- **Sender and receiver QoS selection:** Both senders and receivers should be allowed to change the QoS required by the flow. The sender changes the QoS according to the properties of the data being sent and the receiver may reduce the QoS according to the limitations of the network or workstation at the receiver side.

- **Flow is one-way:** Multimedia data flows in one direction, from sender to receiver, in a single flow. If a reply is required by the receiver, the receiver may elect to use another flow for replying if the reply requires QoS support, or the receiver may use traditional communication if the reply does not include multimedia.

- **Single type of data:** A single flow has a single type of multimedia data flowing at one time. Two different flows are required to transmit a video presentation and an audio song to the same recipient. The same flow may not be used for both transmissions, as the QoS parameters of both types of multimedia data are different. It is also the case for two audio connections as both audio connections may require different QoS parameters.

- **Connection-less communication:** Connection-less communication will be used for sending packets of multimedia data on the underlying network. This allows a better performance by eliminating the overhead of packet re-sending and ordering which are normally performed by connection-oriented transport protocols. The need for packet re-sending and ordering does not exist with multimedia data, since, by definition, multimedia data is not useful if not transmitted on time.

4.2.3 Architecture

QoS architectures are typically composed of three cooperating components: a sender, a receiver and the system layer. The sender is typically an application developed by the user, which requires a QoS controlled channel to send information to the second entity, the receiver. The receiver is also a user-level application that consumes the information it receives. One application may be a sender for some data and a receiver for others, such as a distant learning server that sends video presentations to other destinations and receives their audio questions on another channel. The system layer is the QoS-aware transport protocol that allows QoS specification and maintenance for the flows created by the senders. The proposed work involves a sender-side component and a system-level component. The first component allows user applications to create and manage QoS-capable flows, whereas the latter is responsible for the internal maintenance, management of flows and the prioritization of flow data. Figure 13 shows the main component-interaction diagram for the proposed work.

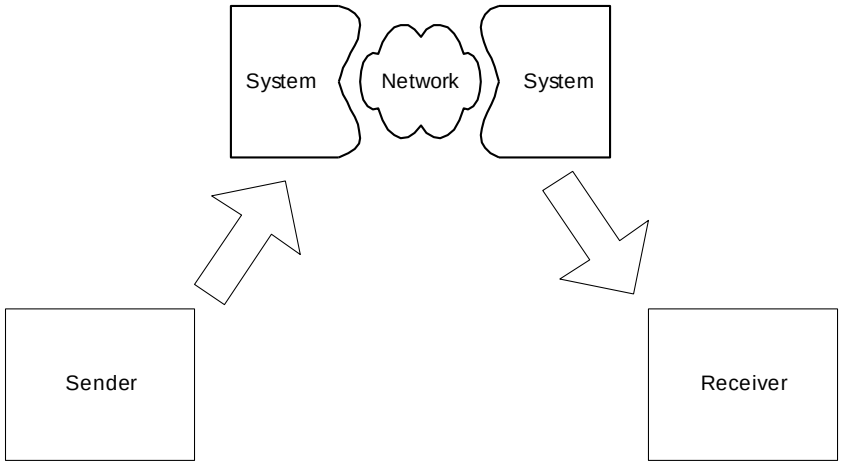


Figure 13 - Main System Component Interaction Diagram

The proposed work does not include any receiver-side components. The receiver is assumed the same for QoS-aware flows and traditional burst data. A receiver that was designed to receive multimedia data in burst mode will be the same as the one that receives controlled flow data. All QoS management functions can be performed at either the sender-side or the system level without the need for the interference of the receiver.

The system-level component is made up of two main components: the QoS Daemon (MQOSD) and the Realtime Scheduling Daemon (MRTSD). The MQOSD is responsible for receiving the requests for creating new flows. It allows for QoS specification and performs the required QoS mapping to translate user-level QoS parameters to network-level QoS parameters. The MQOSD is also responsible for the initial admission control for the new flows. Furthermore, the MQOSD stores information related to every flow in the system. The information stored includes

required QoS parameters as well as the current attainable QoS levels for every flow. The MQOSD performs QoS notification when the attainable QoS levels do not match the levels agreed upon in the specified QoS.

The MRTSD is responsible for the transmission of flow packets. The MRTSD receives the data to send from user applications and creates deadlines for them according to the QoS data for the flow stored in the MQOSD. The MRTSD calculates a deadline for every packet and performs priority scheduling of packets in order to dispatch them in a manner that meets the QoS specified by each flow in the MQOSD. The MRTSD performs QoS monitoring operations for every flow and reports its results to the MQOSD for storage with the flow information. QoS monitoring is performed by sending out-of-bound packets to the remote MQOSD and measuring the required QoS parameters on these packets.

The MQOSD and the MRTSD processes are designed to exist once per workstation that participates in the QoS managed system. The MQOSD for the workstation stores information relating to all flows whose senders originate from this same workstation. The MRTSD for the workstation dispatches and monitors the packets of the flows whose QoS data are stored in the MQOSD of the local workstation. The proposed work assumes that, in a LAN, the monitoring information collected by a single MRTSD is representative of the information collected by other MRTSD processes in the LAN. Consequently, MRTSD processes in a LAN are not required to communicate together to form a complete picture of the LAN utilization. Figure 14 shows the detailed component interaction diagram for the proposed system.

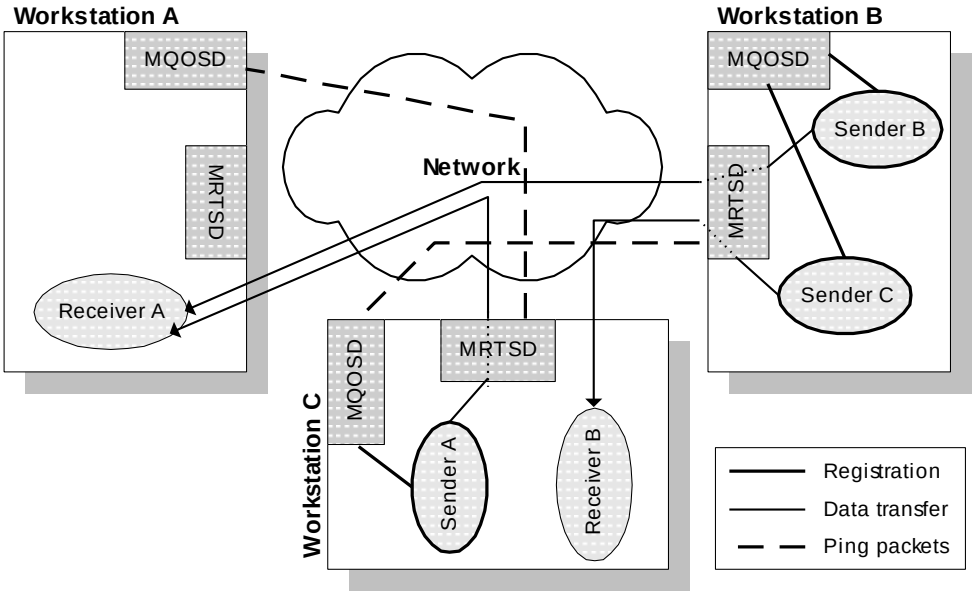


Figure 14 - Detailed Component Interaction Diagram

4.2.4 QoS Specification and Mapping

Before a sender is allowed to send QoS controlled data, it must register a flow for the stream of data to be sent. During flow creation, the sender specifies the QoS requirements for the data. The specified QoS level is registered as a contract between the user application and the system. The system guarantees that the required QoS will be provided, or a notification will be sent if the registered QoS level cannot be kept. The system, also, uses the specified QoS in all the calculations that it performs on the packets belonging to this flow.

The present work adopts a QoS specification set that directly matches the need of multimedia applications. A multimedia application can send either *audio* or *video* data. For both types of data, *threshold values* are specified to indicate the QoS requirements for the different parameters. For audio types, the application needs to specify the frequency of sampling, number of bits per sample, and the number of

audio channels. Video flows have to specify the horizontal and vertical resolution, number of colors per pixel, and number of frames per second of display. Both audio and video flows need to specify the average used compression ratio in order to estimate the throughput requirements of the flow correctly. All the preceding parameters are used to calculate a *lower threshold* for the throughput required by the flow. Interactivity can be specified by flows to indicate whether they require HIGH or LOW interactivity. Interactivity measures are required to estimate the level of synchronization required between the sender and the receiver. The tolerance level, also, needs to be defined for every flow. The tolerance level is the level of acceptable deviation from the specified threshold values. Figure 15 shows the list of QoS parameters required for the creation of a flow.

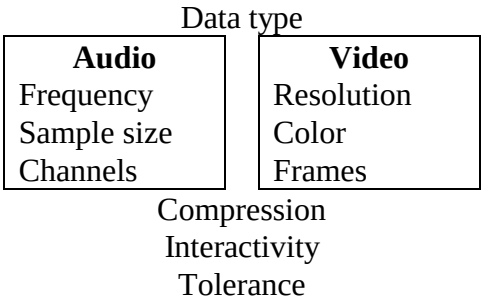


Figure 15 - QoS Specification Parameters

The QoS specification process begins at the sender-side components, where the sender uses these components to specify its QoS requirements. The sender-side components interact with the local MQOSD which maps the QoS request into network QoS parameters to determine the availability of the requested QoS. The MQOSD replies with an accept/deny result to the sender through the sender-side components.

QoS mapping is performed at the local MQOSD by calculating network-level parameters from the specified user-level QoS. The network-level parameters of relevance to continuous media are *throughput, end-to-end delay and error rate*. Throughput is the bandwidth to be used by the continuous media data in order to be transferred to the receiver on time. This specifies the amount of data flowing through in a certain period. Throughput is important because it provides the system with an indication on how much of the shared network bandwidth is required by each flow. End-to-end delay is the amount of time spent from when the data was ready for transmission at the sender until the data was ready for consumption at the receiver. The importance of end-to-end delay is that it directly affects the interactivity between the sender and the receiver and consequently it affects the human perception of the continuous media sent in a flow. The error rate is the ratio of packets that may be transmitted at a lower QoS level. The human perception of continuous media transmitted varies inversely with the loss rate. Together, throughput, end-to-end delay, and loss rate define the behavior of a flow and the acceptable level of quality that is allowable for correct perception at the receiving side. The three parameters also define the aspects of the network that affect the level of quality of the transmitted continuous media.

QoS mapping is performed by mathematically deriving the three network parameters from the user-level QoS parameters requested during QoS specification. Throughput calculation depends on the type of media specified. Throughput is specified in bytes per second.

For audio flows:

Equation 1:

$$\text{Throughput} = frequency \times channels \times \frac{samplesize}{8} \times \frac{1}{compression}$$

For video flows:

Equation 2:

$$\text{Throughput} = xresolution \times yresolution \times \frac{\lceil \log_2(colors) \rceil}{8} \times frames \times \frac{1}{compression}$$

The end-to-end delay parameter is derived from the interactivity parameter of the user-level QoS. End-to-end delay is specified in milliseconds. The user-level interactivity parameter was assumed for simplicity to be one of two values: either high or low. The mapping mechanism assumes a hard-wired value of 500 ms for high interactivity and 1000 ms for low interactivity. The error rate parameter is the same as the user-level tolerance parameter. Error rate is measured in percentage of packets that do not arrive at the receiver side. This may be due to either network error, packet loss, or packet dropping due to buffer overflows.

4.2.5 QoS Maintenance

In order to allow a sender to specify flow parameters using QoS specification, the system must provide an interface to its senders. This interface is provided by the sender-side components mentioned in section 4.2.3. The sender interface is responsible for granting the sender access to all the functions of the QoS maintenance subsystem. The QoS maintenance subsystem provides the following functions:

- **Flow creation:** establishing a dedicated connection to a specific receiver
- **Sending data:** sending data packets to the flow receiver
- **QoS selection:** altering the contracted QoS level
- **Flow termination:** close connection with the flow receiver

Flow creation starts by a request from the sender to the local MQOSD to create a flow with a specified QoS. The local MQOSD evaluates the request and returns the result of the QoS admission process back to the sender. The local MQOSD stores the contracted QoS level together with the flow information for later usage during QoS management. Each local MQOSD stores the contracted QoS levels for all the senders on its workstation.

After the flow is created, the sender may send continuous media on the flow by submitting the packets to the sender-side components. The packets are then forwarded to the local MRTSD, which stores the packet together with the identifier of the flow sending in a priority queue for scheduling and delayed transmission. The local MRTSD contacts the local MQOSD to retrieve the QoS level required by the flow to which the packet belongs. This allows the MRTSD to take smart decisions as to how to schedule the packets. The MRTSD schedules packets according to a realtime-scheduling algorithm. When a packet is ready for transmission, the MRTSD contacts the receiver and sends the packet directly using the underlying transport protocol. The MRTSD performs no checking to ensure that the packet has reached its destination correctly.

The MRTSD uses an Earliest-Deadline-First realtime-scheduling algorithm to prioritize the packets from different flows. A deadline is calculated for every packet using the following equation:

Equation 3: $$t = \begin{cases} t_0 & \text{, for } Q_a = 0 \\ t_0 + \frac{s}{Q_d} - \frac{s}{Q_a} & \text{, for } Q_a \neq 0 \end{cases}$$

Where:

t: the deadline for sending the packet

t_0: time when the packet was submitted for sending

s: the size of the packet in bytes

Q_d: the desired throughput as per the QoS contract for the flow

Q_a: the actual network throughput as measured in QoS monitoring

An actual throughput (Q_a) of zero denotes that no measurements have been taken for the network yet. `The term (S / Q_d) represents the maximum time allowed until the packet to reaches its destination as specified in the QoS level of the flow. The term (S/Q_a) denotes the time required for sending the packet using the network parameters that were measured earlier during QoS monitoring. If the deadline is zero or negative, the packet should be sent immediately. A negative deadline implies that the packet will be late for its recipient. Corrective action for the late packets will be taken during QoS monitoring.

During flow operation, the sender may wish to change the contracted QoS level. The sender may elect to increase the required QoS level to send more detail or may wish to decrease the level in order to release some burden from the network. The sender contacts the local MQOSD with a Select-QoS request. The local MQOSD evaluates the new request and performs the QoS admission process again. The new QoS level is stored with the flow data in the local MQOSD.

After the sender has finished transmitting all the data it requires, it should release the flow by sending a termination request to the local MQOSD. All termination requests are instantly accepted by the local MQOSD and all the flow information is marked for removal once all its pending packets have been sent. Figure 16 shows the typical operation of the QoS maintenance system.

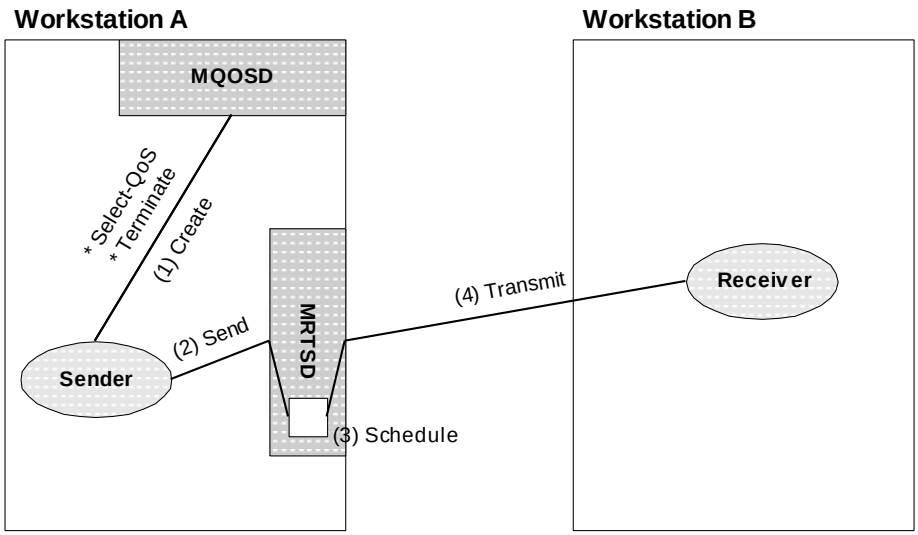


Figure 16 - Typical QoS Maintenance System Operation

4.2.6 QoS Monitoring and Notification

QoS monitoring is performed on every flow during normal flow operation to ensure that the QoS management system adheres to the contract signed with the sender during QoS specification. When the contracted QoS level cannot be maintained, the QoS monitoring subsystem notifies the sender of the QoS degradation in order for the sender to take corrective action. This allows the various senders in the system to cooperate in making room for each other during times of congestion. It also allows flows to terminate if the contracted QoS level cannot be kept.

The QoS monitoring subsystem adopts the packet injection technique in measuring flow performance. Monitoring is performed by sending probe out-of-bound packets together with flow data. The ratio of probe packets to the actual data is determined by the system-level *monitoring rate* parameter. A monitoring rate of 0.1 denotes that 10% of all flowing data on the network is probe data. The scheduler of the MRTSD is responsible for injecting probe packets together with the data packets at the specified monitoring rate. The probe packets are sent out-of-bound to the MQOSD of the receiver instead of being in-bound directly to the receiver. This has the advantage of reducing the overall end-to-end delay of the QoS maintenance subsystem by not introducing further processing at the receiver side. The disadvantage is that more processing needs to be performed at the MQOSD to allow for processing and returning probe requests. Probe packets are used to measure all network QoS parameters: end-to-end delay, throughput and error rate. Monitoring is performed independently on every MRTSD. The various MRTSD in the network do not communicate to share monitoring information. It is assumed that over a large network,

the senders and receivers will be evenly distributed and hence measuring network performance will be similar from any point in the network.

End-to-end delay is measured by calculating the roundtrip delay of a null packet. A null probe packet is transmitted from the local MRTSD to the remote MQOSD. The packet is first delayed by the sender transport protocol, which adds some system delay (D_s). The packet further undergoes network delay (D_n) before it is ready for consumption at the receiver. The remote transport protocol processes the packet, which adds some system delay (D_r), and makes it available to the receiver. The total end-to-end delay is the aggregate of both the network delay and the system delays. Measuring the end-to-end delay at the remote MQOSD requires that the clocks of both the sender and the receiver be synchronized. This adds to the complexity of the overall system and directly affects the operation of other processes at the remote system. The adopted alternative to clock synchronization is to return the null probe packet with a reply consisting of another null packet. Theoretically, the reply packet should undergo as much delay as the original probe packet. The round-trip delay is now the sum of the delay of the probe packet and the delay of its reply. Consequently, the end-to-end delay is approximately half the round trip delay. End-to-end delay calculation is illustrated in Figure 17 and Equation 4. The equation assumes that the time taken in processing the probe packet at the MQOSD and initiating a reply is negligible.

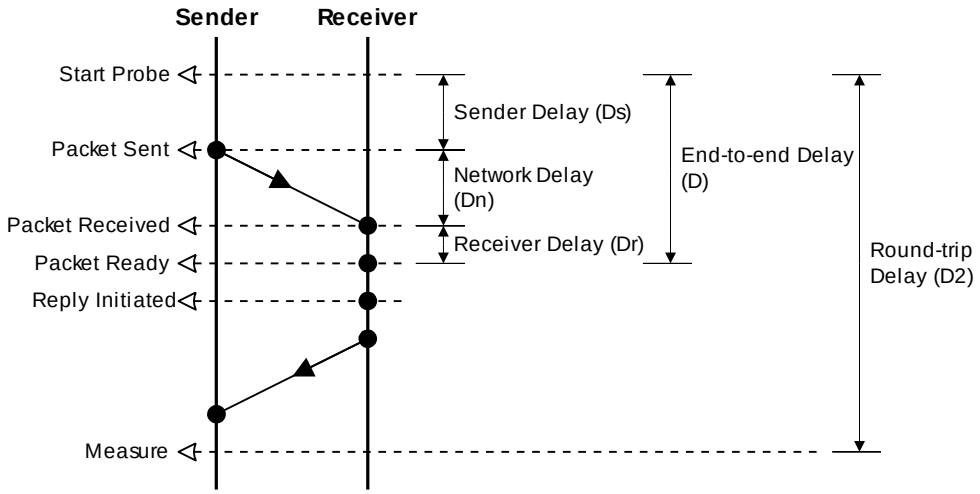


Figure 17 - End-to-End Delay Calculation

Equation 4: $D = D_s + D_n + D_r \approx \frac{D_2}{2}$

Throughput is measured by sending a stream of probe packets and computing the difference between the time of arrival for the first packet and the time of arrival for the last packet. The local MRTSD sends a known number of packets (N_P), which is set by a system parameter, to the remote MQOSD with the rate specified by the measuring rate system parameter. The size of the probe packet (S) is determined by the measured average packet size for the flow. The remote MQOSD starts counting the time as soon as the first packet is received (t_a). Time calculation is stopped when a certain number of probe packets are received. An assumption is made for the upper bound on the network loss rate (denoted L_u). The remote MQOSD stops the time count at time t_b, when $L_u \times N_P$ probe packets are received. This ensures that the MQOSD will not wait indefinitely for a probe packet that was lost. When enough probe packets are received, the remote MQOSD calculates the actual throughput (Q_a) using the following formula:

Equation 5: $$Q_a = \frac{L_u \times N_P \times S}{t_b - t_a}$$

The remote MQOSD composes a reply packet and replies to the MRTSD of the sender with the measured throughput. Figure 18 shows an illustration for the throughput calculation process.

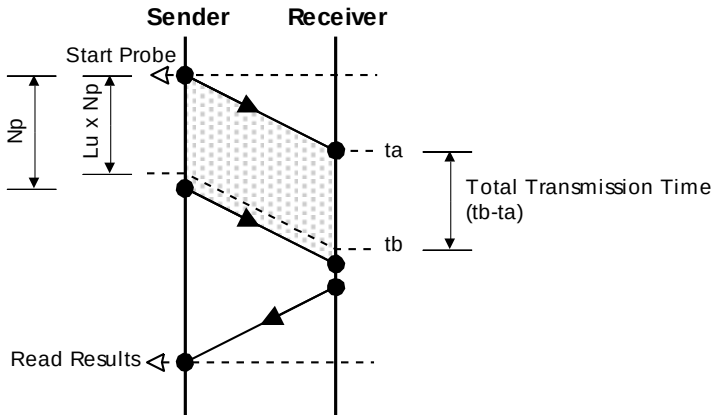


Figure 18 - Throughput Calculation

Error rate is measured by computing the difference between the aggregate of the desired throughput (Q_q) for all flows that share the same network and the actual measured throughput (Q_a) of that network during monitoring. In the proposed design, the assumption of a single LAN is made and hence all flows (N_F) are counted in the sum of desired throughput. The error rate is calculated using the formula shown in Equation 6.

Equation 6: $$E = \frac{\left[\sum_{i=1}^{N_F} Q_q(i)\right] - Q_a}{\sum_{i=1}^{N_F} Q_q(i)}$$

A negative error rate means that the network can provide a higher throughput than what it is asked to grant. This means that there is no error (error rate is zero).

Monitoring results for every flow are eventually stored in the local MQOSD for the sender of every flow. End-to-end delay is calculated at the local MRTSD and then reported to the local MQOSD. Throughput is calculated at the remote MQOSD and then sent back to the local MRTSD, which, in turn, reports it to the local MQOSD. Error rate is calculated at the local MQOSD because it requires knowledge about the desired throughput of all local flows.

The local MQOSD keeps history of previous monitoring results rather than saving the only the latest values. This has the advantage of averaging the monitoring results and thus excluding any spontaneous and temporary spikes in the results. This reduces any unnecessary notification of low performance that may have lasted for only a short period. The ratio of new to old data to keep is defined by the *History Ratio* system parameter. Equation 7 shows the formula for calculating the new values while keeping history of the old values.

Equation 7: $X_{current} = \beta X_{old} + (1 - \beta) X_{new}$

Where:

$X_{current}$: the value (delay, throughput or error rate) to store

X_{old}: the old stored value for delay, throughput or error rate

X_{new}: the current measured value for delay, throughput or error rate

β: the history-ratio system parameter

After every new set of measured network parameters is calculated, the MQOSD compares the calculated measured QoS with the contracted QoS to check for performance degradation. If the calculated throughput is lower than the desired throughput by an amount greater than the error rate, the sender is notified of degradation in QoS level. Similarly, if the calculated delay is larger than the desired delay, the sender is notified of the degradation in QoS level. The sender is expected to, but without any obligation, to take corrective action that would either reduce the requested QoS-level or terminate the flow due to insufficient resources.

4.2.7 QoS Admission

The QoS admission process attempts to perform a pre-entry QoS assessment for flows prior to flow creation at the local MQOSD. The QoS level specified by a flow during QoS specification is assessed against the current network status before deciding whether to admit the flow into the system. A flow will be allowed access only if the system "believes" that it can provide the desired QoS-level. The QoS admission subsystem uses fuzzy logic and a neural network to make a sensible decision.

QoS admission is performed on a flow that has a desired QoS level with a desired throughput (Q_q), delay (D_q) and error rate (E_q). The QoS admission subsystem also utilizes the current network status, which provides information on the actual measured throughput (Q_a), delay (D_a) and error rate (E_a). The system uses the preceding six parameters as inputs to the QoS admission process. The output of the process is a single decision to either accept or deny the creation of the flow.

The QoS admission subsystem is composed of two main components: the fuzzy network that assesses the inputs and the neural network that produces a decision. The

fuzzy network employs fuzzy logic to compare the requested parameters with the measured actual network parameters to produce an intermediate “acceptability” level for these parameters. The acceptability level of each parameter describes the extent to which the network can satisfy the requested parameter. A higher acceptability means that the network is more likely to meet the requested parameter. Three acceptability levels are produced from the fuzzy network for the three network parameters: delay, throughput and error rate. The acceptability levels are fed as inputs to a neural network that produces the final decision. The neural network combines the three acceptability levels and produces a sensible decision to accept or deny the flow creation request based on the relative importance of each of the three parameters. Figure 19 illustrates the details of the QoS admission subsystem.

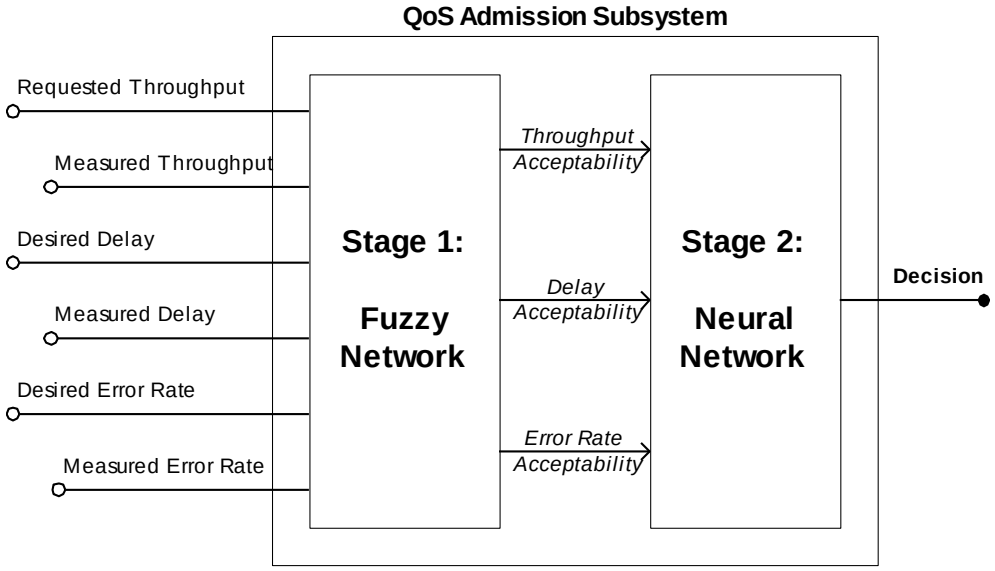


Figure 19 - QoS Admission Subsystem

The fuzzy network uses the following algorithm to produce the acceptability levels:

1. Obtain the average network status from the average of the monitored results for all flows stored at the local MQOSD.

2. Normalize the network status parameters in the range [0-1]. Use the normalization formulas shown in Equation 8, Equation 9 and Equation 10.

Equation 8: $$\text{Throughput} = \frac{\left[\sum \text{requested throughput}\right] - \text{measured throughput}}{\sum \text{requested throughput}}$$

Equation 9: $$\text{Delay} = \frac{\text{measured delay}}{\text{maximum requested delay}}$$

Equation 10: $$\text{Error rate} = \text{measured error rate}$$

3. Fuzzify the network status using the fuzzification graph shown in Figure 20.

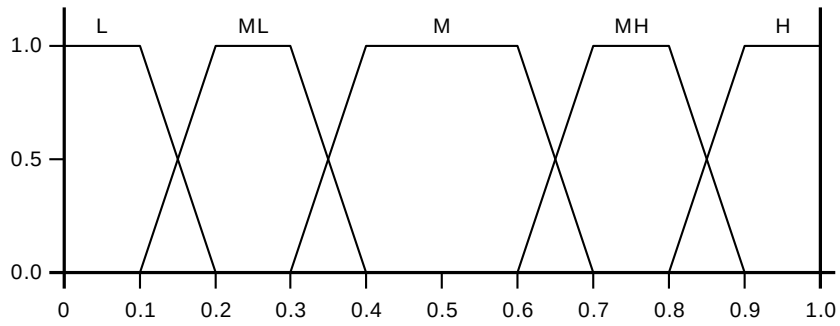


Figure 20 - Fuzzification and Defuzzification Graph

4. Normalize the user request in the range [0-1] using the normalization formulas shown in Equation 11, Equation 12 and Equation 13.

Equation 11: $$\text{Throughput} = \frac{\text{requested throughput}}{\text{average measured throughput} \times 1.5}$$

Equation 12: $$\text{Delay} = \frac{\text{maximum requested delay} - \text{measured delay}}{\text{maximum requested delay}}$$

Equation 13: $$\text{Error rate} = 1 - \text{measured error rate}$$

5. Fuzzify the user request using the fuzzification graph shown in Figure 20.

6. Use the fuzzy decision table shown in Table 1 to get fuzzy values for the acceptability levels. A higher fuzzy value denotes a higher potential for acceptance by the network.

Table 1 - Fuzzy Decision Table

		Requested Value for Parameter				
		L	**ML**	**M**	**MH**	**H**
Measured Value for Parameter	**L**	H	H	H	H	H
	ML	H	H	MH	M	M
	M	H	MH	M	M	ML
	MH	MH	M	M	ML	L
	H	M	ML	ML	L	L

Where: (H= high; MH= medium high; M= medium; ML= medium low; L= low)

7. Defuzzify the acceptability levels to get crisp values for the acceptance potential in the range [0-1] using Figure 20.

8. Feed the acceptance levels to the next stage for obtaining a single decision.

The next stage is the neural network phase. The neural network takes as input the acceptability potentials for the three network parameters and produces one output which is the acceptance decision (YES or NO). The neural network is a feed-forward neural network with one hidden layer containing three nodes and one output layer containing one node. The proposed network is shown in Figure 21. The network is trained using the back-propagation algorithm using the training set shown in Appendix A.

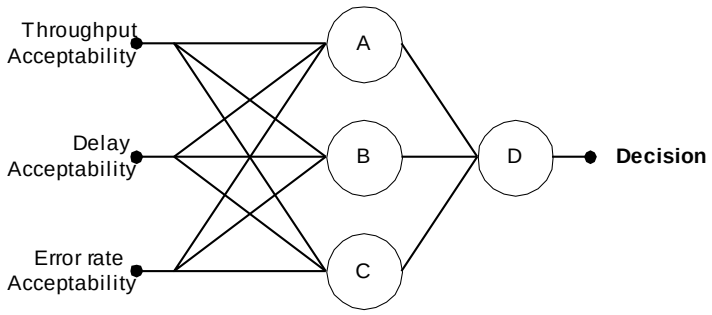


Figure 21 - QoS Admission Neural Network

4.3 Prototype

A prototype system was developed to verify the design and simulate its functionality. The prototype is also used to measure results for the effectiveness and performance of the proposed system. The prototype is composed of a library and a high-level language interface that provide senders with access to QoS functions of the proposed

QoS management system. The prototype also includes two system applications that represent the MQOSD and the MRTSD, which should be run on every host participating in the QoS-managed network.

The developed prototype has the limitations of running on a single LAN as opposed to a WAN that is a necessity for real-life multimedia applications. The LAN assumption was made to conform to the assumptions made earlier in the system design. The prototype is also limited to the implementation of unicast flows instead of multicast flows that are commonplace in multimedia applications. The design of the prototype allows flows to use multicast facilities but the implementation was limited to unicast for simplicity.

4.3.1 Platform

Object-oriented design (OOD) was employed in the course of designing the prototype system. Object-oriented design was mainly selected because it allows for simple additions to the design and for providing placeholders for extras that need to be added to the design later. OOD is also useful due to its readiness to easily reuse design components.

The prototype system is implemented in C++ on the Linux platform. The prototype is compiled with Gnu C++ on Red-Hat Linux. The developed prototype completely runs in user-mode and does not modify any system-level operating system components. The system builds on TCP/IP's UDP protocol to provide the transport protocol functionality. The developed system is built on top of UDP to change its interface and add to its functionality. Inter-process communication (IPC) is employed in all communication within a single host for optimization.

The prototype is designed using the Booch object-oriented design methodology [Booch 94]. The important design elements of the Booch methodology are the class diagram and the object diagrams. The class diagram identifies the main system components and their interaction. The object diagrams identify the main scenarios of system operation and shows how instances of the class components would interact during each scenario.

4.3.2 Class Diagram

The main classes of the prototype are the *SenderFlow*, *MQOSD*, and *MRTSD*. These represent the sender-side interface, the MQOSD process and the MRTSD process, respectively. Figure 22 shows the class diagram for the prototype system. Class details and member functions are listed in Appendix B.

The main interface to the sender application is the *SenderFlow* class. The sender should create an instance of this class for every flow it wishes to create. The class contains member functions for flow creation and termination, QoS selection as well as data sending. The sender interacts with the *Buffer* class to send data using the flow. The *Buffer* class provides easy handling of data for simple sending of different data types. The *SenderFlow* can only send data encapsulated in a *Buffer* object. The *SenderFlow* itself is a specialization of the *Flow* class, which is an abstract class providing general flow functionality. The *Flow* class may be used as a base class for further classes that provide enhanced system functionality and/or a different user interface to the sender. This may also be useful in providing a class that allows multicast group membership and handling.

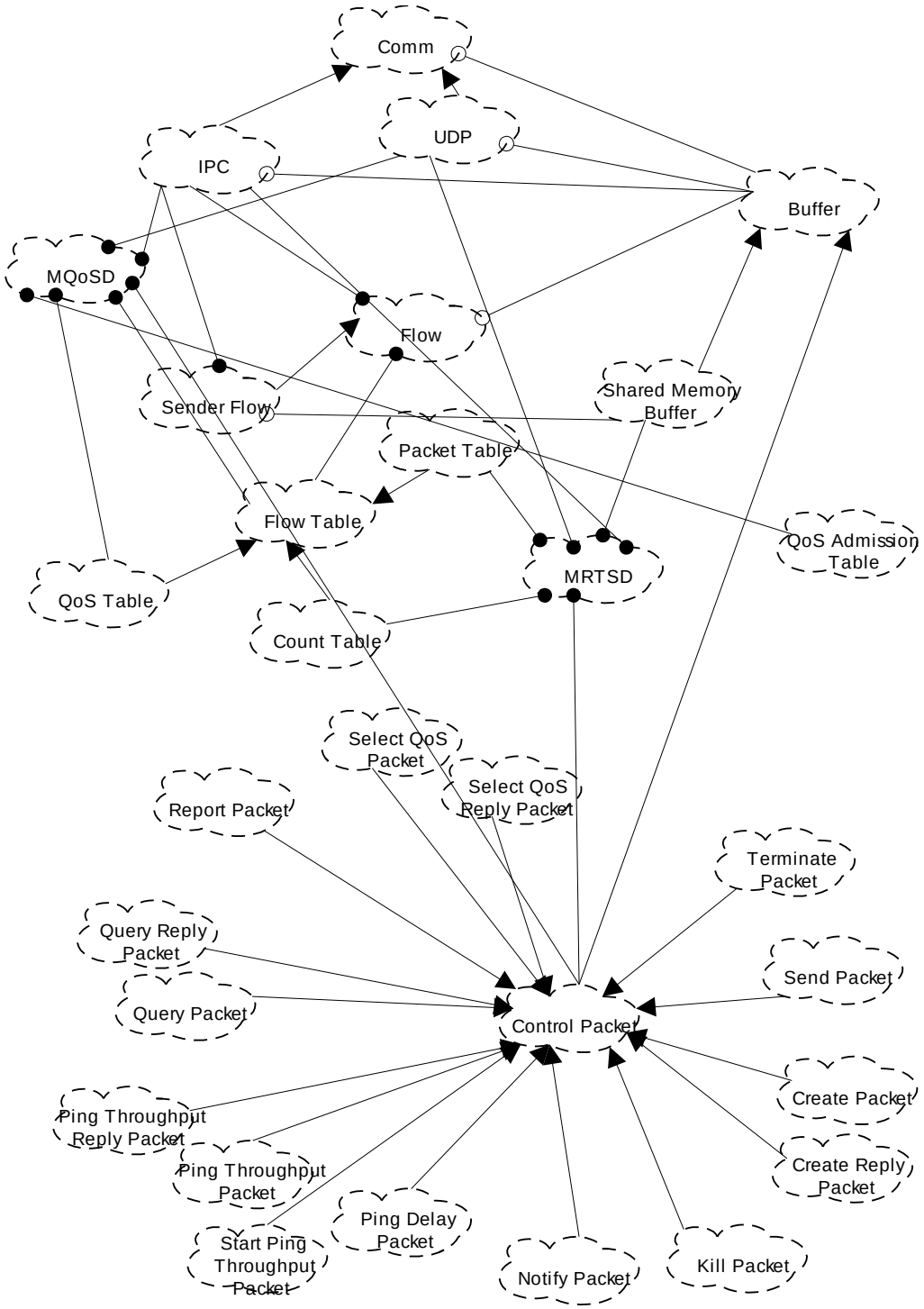


Figure 22 - Class Diagram for Prototype

The *SenderFlow* uses the *IPC* class for interacting with the local MQOSD and MRTSD processes. The *IPC* class is a specialization of the *Comm* class, which

provides general communication capability. The *IPC* class interacts with *Buffer* classes to extract the data to send to the MQOSD. Data sent to the MRTSD are contained in a *SharedMemoryBuffer* class, which uses shared-memory IPC to minimize the memory copying operations in order to enhance the overall system performance and reduce sending delays. *SharedMemoryBuffer* is a specialized *Buffer* that uses shared memory IPC storage instead of main memory storage.

The *MQOSD* class provides the functionality of the MQOSD process as described earlier in the proposed design. The *MQOSD* waits for control packets from any sender to provide QoS management functionality. It acts as the main storage and QoS decision-maker on every host. The *MQOSD* uses a *ControlPacket* class to receive and identify any control packets it receives. It also contains a *QoSTable* that stores all information related to the flows created on its host. The *QoSTable* is a descendant of the *FlowTable* class, which is a parent class that provides generic indexing on flow identifiers. The *MQOSD* utilizes the *IPC* class for inter-process communication with senders to receive flow management requests from the *SenderFlow* classes. The *MQOSD* also uses the *UDP* class, which allows UDP communication over the network with remote MRTSD processes. The *UDP* class is another descendent of the *Comm* general communication class. The *MQOSD* also contains a *QoSAdmissionTable* class that stores the QoS admission-control table that uses fuzzy logic to take admission control decisions.

The *MRTSD* class provides the functionality for the MRTSD process, which is to send flow data using the specified QoS levels. The *MRTSD* interacts with the local *MRTSD* using *IPC* classes and with the remote MRTSD processes using *UDP* classes. The

MRTSD receives data to be sent in *ControlPacket* classes that contain references to the original data in stored in *SharedMemoryBuffer* classes. The *MRTSD* stores the packet in a *PacketTable* prior to sending them. Packets in the *PacketTable* are first prioritized according to QoS-driven deadlines. The *MRTSD* contains an Earliest-Deadline-First scheduler that schedules and sends the packet while the *MRTSD* is receiving control packets from senders. The *PacketTable* class is another descendant of the *FlowTable* class because it is indexed on the flow identifier. The *MRTSD* contains a *CountTable*, which contains statistics for the data sent by each flow. This allows for the calculation of the time of sending probe packets. Probe packets are sent as *ControlPacket* classes using *UDP* to the remote *MQOSD* processes.

Various specialized classes of the *ControlPacket* class exist to provide the out-of-bound control functionality. *CreatePacket*, *CreateReplyPacket*, *TerminatePacket*, *SelectQoSPacket* and Select*QoSReplyPacket* are used by the *SenderFlow* and the *MQOSD* to manage QoS flows. *SendPacket* is used by the *SenderFlow* and the *MRTSD* to send flow data. *PingDelayPacket*, *StartPingThroughputPacket*, *PingThroughputPacket*, and *PingThroughputReplyPacket* are used to perform monitoring operations by both the *MRTSD* class and the remote *MQOSD* processes. *ReportPacket* is used by the *MRTSD* to report results of the monitoring process. *NotifyPacket* is used by the *MQOSD* and the *SenderFlow* to notify senders of QoS degradation. *QueryPacket* and *QueryReplyPacket* are used by the *SenderFlow* and the *MQOSD* to inquire about the current operating QoS level for a flow. *KillPacket* is used by the system developer to gracefully terminate *MQOSD* and *MRTSD* processes prior to system shutdown.

4.3.3 Scenarios

The typical scenarios in the proposed system are those involving QoS management functions. The operation of these functions exhibits the most important object interactions. These functions are:

- Flow creation
- Data sending
- Realtime packet scheduling
- QoS monitoring
- QoS degradation
- QoS selection
- Flow termination

4.3.3.1 Flow Creation

Flow creation occurs when a sender wishes to establish a new multimedia data flow. It involves QoS specification by the sender and QoS admission by the MQOSD. The result of this process is either the denial of service or the creation of a flow and the storage of its parameters in the MQOSD. Figure 23 shows the object diagram for a typical flow creation process that results in acceptance.

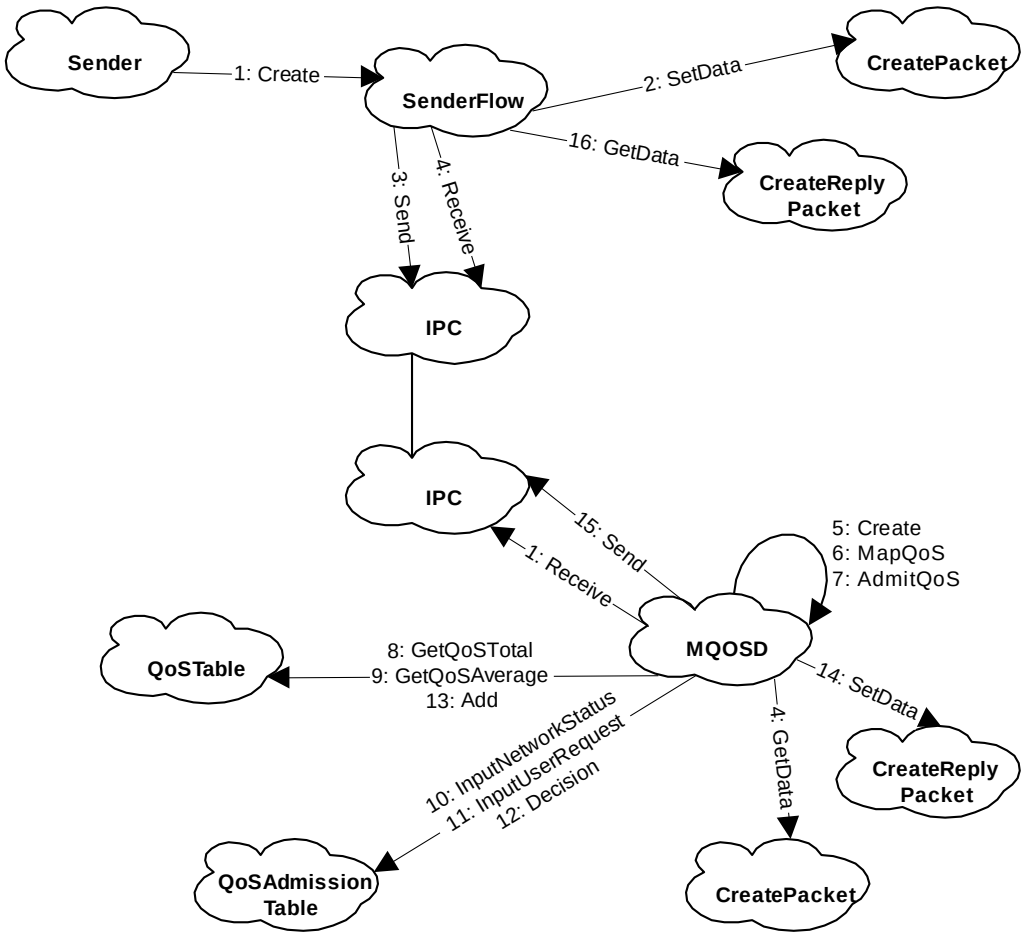


Figure 23 - Flow Create Object Diagram

Flow creation starts with a Create request from the sender to the SenderFlow. Meanwhile, the MQOSD is already ready for receiving data from any sender. The SenderFlow sets the specified QoS data into a CreatePacket and sends it using IPC. The MQOSD receives the request using IPC and gets the data from the CreatePacket. It identifies the data as a Create request and issues a Create operation in the MQOSD. The operation proceeds by mapping the specified QoS into network parameters and then performing QoS admission on the mapped parameters. The admission control mechanism gets QoS totals and averages from the QoSTable. The QoS averages are

input to the QoSAdmissionTable together with the user-specified QoS and a decision is requested. When the decision is positive, the flow is added to the QoSTable. Success information is composed in a CreateReplyPacket and then sent back using IPC to the sender. The sender receives the reply and gets the decision from the CreateReplyPacket. The decision is passed back as the result of the Create operation.

4.3.3.2 Data Sending

Data sending is performed by the sender in order to transfer data to the receiver using the specified QoS. It involves scheduling and transmission at the MRTSD. Figure 24 shows the object interactions for a typical send operation.

Data sending starts by the sender inserting data into the shared memory buffer. The sender requests to send the buffer from the SenderFlow. The SenderFlow packs the data into a SendPacket and sends the data using IPC to the MRTSD. Meanwhile, the MRTSD was waiting for a send request from any sender. The MRTSD unpacks the data from the SendPacket and composes a QueryPacket to be sent to the local MQOSD. The local MQOSD receives the QueryPacket and contacts the QoSTable to get flow QoS information. The MQOSD composes a QueryReplyPacket and sends it back to the MRTSD. The MRTSD gets the QoS information from the QueryReplyPacket and inserts the packet into the PacketTable along with its calculated deadline. The CountTable is also incremented for the flow to adjust QoS monitoring counters. Once the packet is in the packet table, it is ready for consumption by the realtime scheduler of the MRTSD.

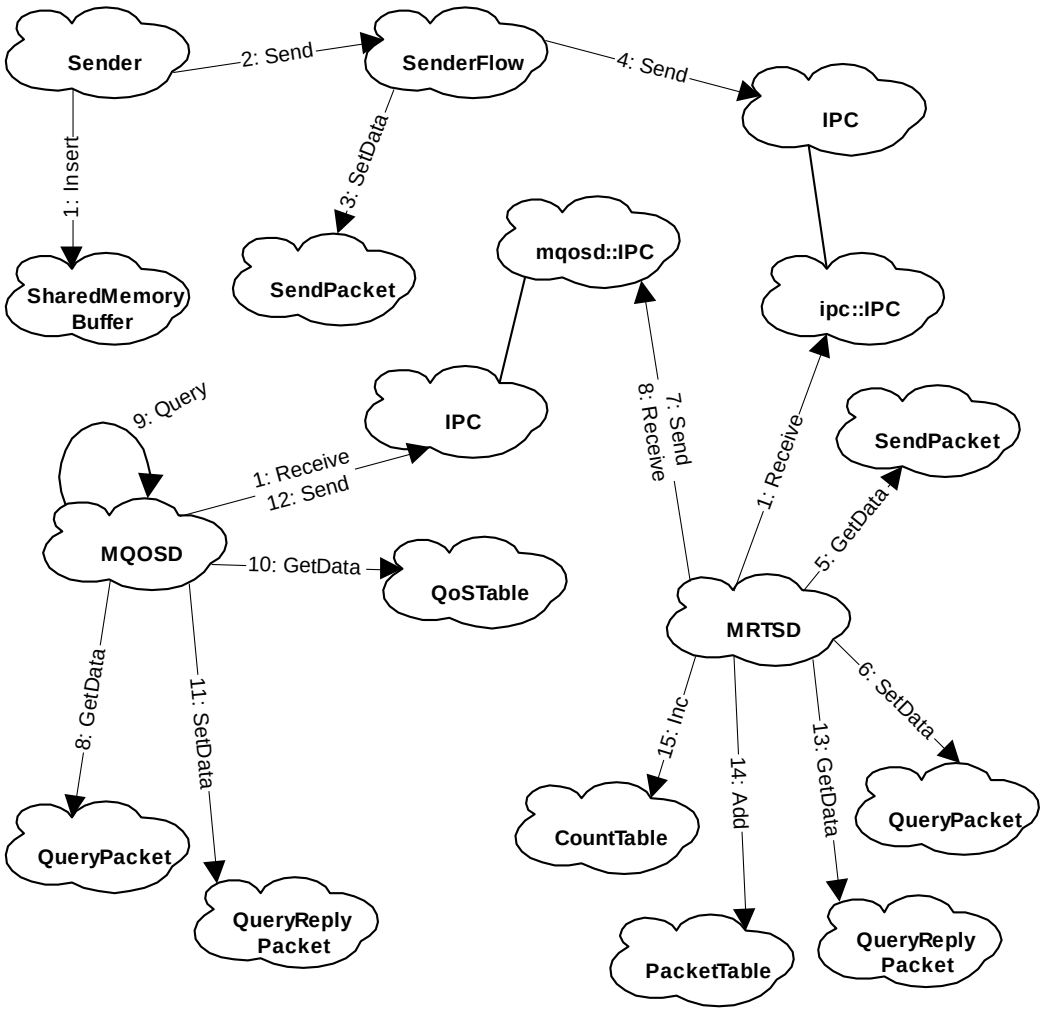


Figure 24 - Data Sending Object Diagram

4.3.3.3 Realtime Packet Scheduling

Realtime packet scheduling is performed whenever the packet queue in the MRTSD is non-empty. The MRTSD utilizes an Earliest-Deadline-First realtime scheduling technique using the deadlines that were set during packet acceptance by the MRTSD. The highest priority packet is extracted and sent to its destination. Figure 25 shows the object diagram for a typical scheduling scenario.

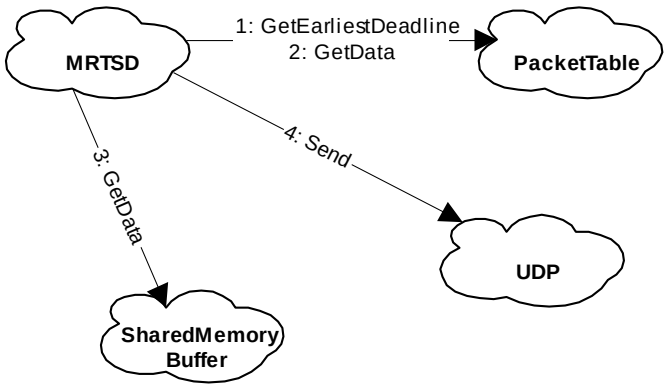


Figure 25 - Realtime Packet Scheduling Object Diagram

The MRTSD scheduler process is synchronized with the MRTSD using semaphores in order to detect any packet arrival. Whenever packets are ready, the MRTSD contacts the PacketTable to get the packet with the earliest deadline. The MRTSD gets the packet data and extracts the shared memory buffer containing the data to send it to the receiver using UDP. The receiver is a normal TCP/IP application executing the *recvfrom* system call to receive UDP packets.

4.3.3.4 QoS Monitoring

QoS monitoring is performed regularly at the specified system rate for all flows. QoS monitoring involves probing for network performance and storage of results at the local MQOSD. Figure 26 shows a typical scenario for QoS monitoring.

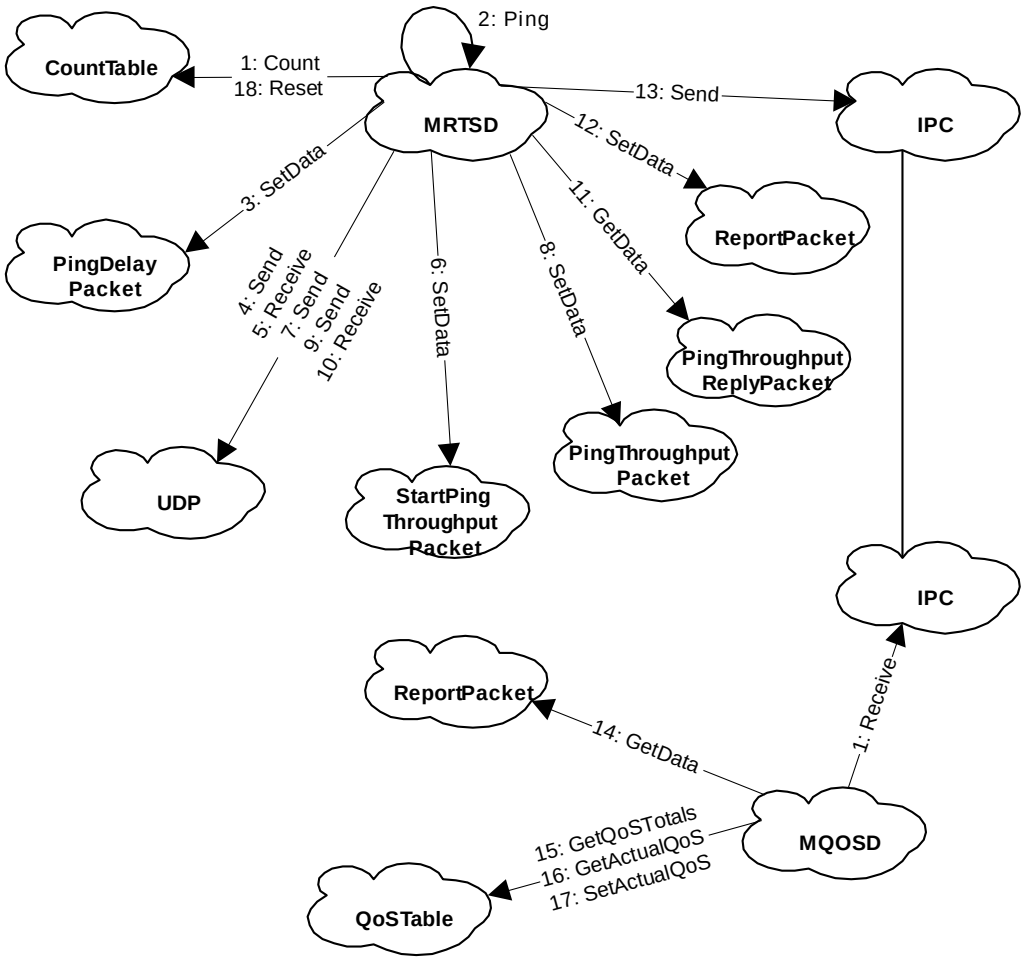


Figure 26 - QoS Monitoring Object Diagram

QoS monitoring starts by the MRTSD scheduler counting the size of the packets sent for the flow. When the threshold is reached, QoS monitoring is initiated. First, the MRTSD creates a PingDelayPacket and sends it using UDP to the remote MQOSD. The MRTSD waits for the results of the PingDelayPacket and starts the process of throughput measurement. The MRTSD sends a StartPingThroughputPacket using UDP followed by multiple PingThroughputPacket datagrams. The MRTSD collects the results in a PingThroughputReplyPacket. After all measurements were made, the MRTSD creates a ReportPacket and sends it using IPC to the local MQOSD. The

local MQOSD receives the packet and unpacks the ReportPacket. After that, the MQOSD retrieves old QoS measurements for the flow from the QoSTable and stores the newly calculated measurements in the QoSTable.

4.3.3.5 QoS Degradation

QoS degradation occurs when the local MQOSD detects that the QoS measurements being stored for a flow are lower than the contracted QoS level. QoS degradation notifies the sender of the situation in order to take appropriate corrective action. Figure 27 shows the object diagram for a typical QoS degradation scenario.

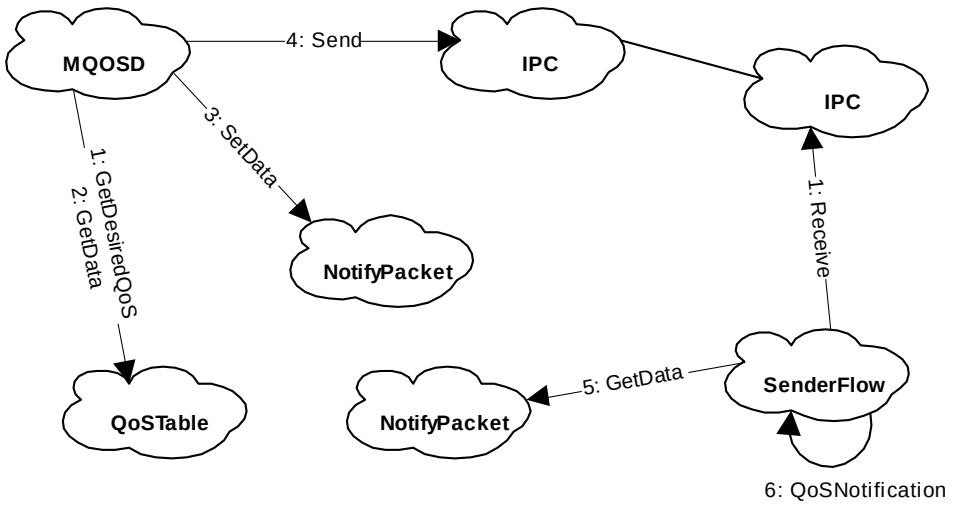


Figure 27 - QoS Degradation Object Diagram

When the MQOSD gets the desired QoS level for the flow, it compares it with the measured values obtained during QoS monitoring and reporting. If the measured level is lower than requested, the MQOSD composes a NotifyPacket and sends it using IPC to the SenderFlow. The SenderFlow is already waiting for any notifications in a separate QoSNotificationHandler process. The SenderFlow reads the contents of the

NotifyPacket and executes the overloaded QoSNotification member function. Sender applications override the QoSNotification member function to achieve custom functionality upon QoS degradation notification.

4.3.3.6 QoS Selection

QoS Selection is performed at the sender's request. A sender may wish to perform QoS selection when the contracted QoS levels need to be altered. The QoS selection process involves QoS specification and QoS admission (readmission). Figure 28 shows the object diagram for a typical QoS selection request.

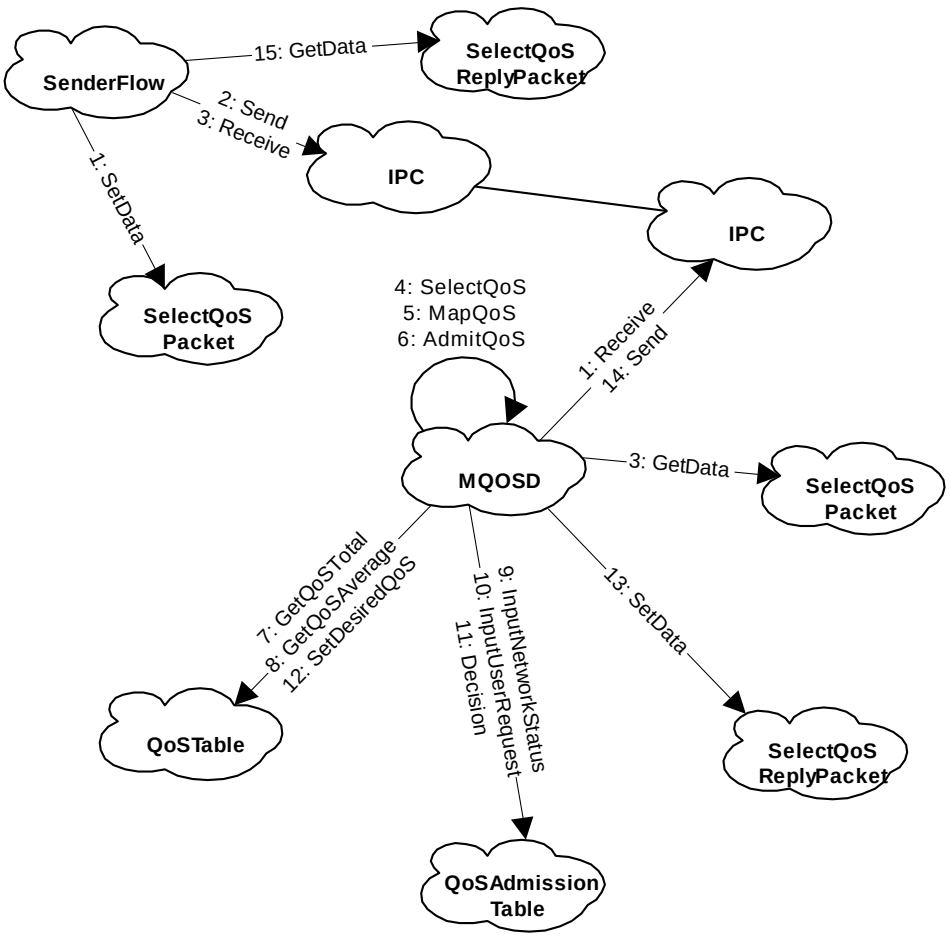


Figure 28 - QoS Selection Object Diagram

The sender initiates the process by requesting QoS selection from the SenderFlow. The SenderFlow, in turn, creates a SelectQoSPacket and sends it through IPC to the local MQOSD. The local MQOSD receives the data, unpacks the SelectQoSPacket, and invokes the local QoS selection code. The QoS selection process involves QoS mapping and QoS admission similar to that performed during the flow creation process described in section 4.3.3.1. The result of QoS admission is stored in the QoS table for later usage. The result is then packed in a SelectQoSReplyPacket and sent back to the SenderFlow using IPC. The SenderFlow gets the result from the SelectQoSReplyPacket and passes it to the sender.

4.3.3.7 Flow Termination

The sender chooses to terminate a flow when it has finished sending all the data in the flow. Figure 29 shows a typical flow termination scenario.

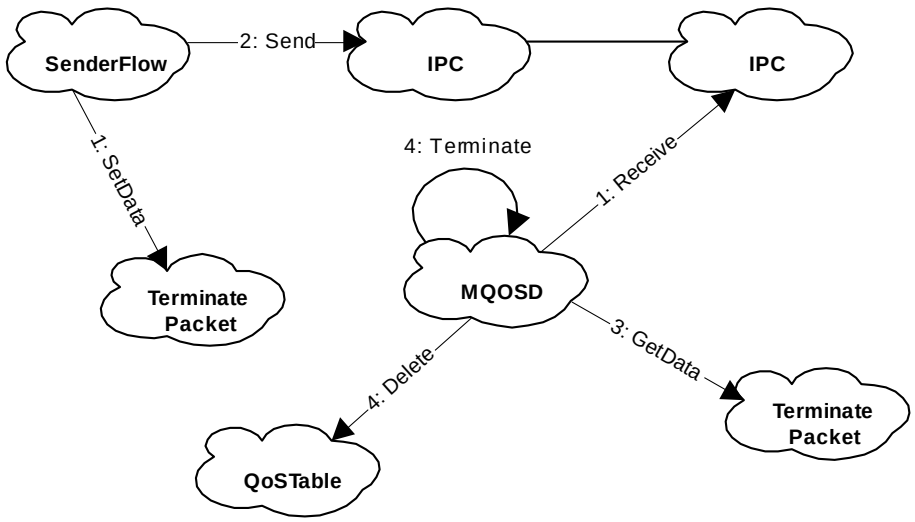


Figure 29 - Flow Termination Object Diagram

The SenderFlow creates a TerminatePacket and sends it using IPC to the MQOSD. The MQOSD decodes the packet and executes the local terminate procedure. The procedure removes all the deleted flow's data from the QoSTable.

Chapter 5 EXPERIMENTAL RESULTS

A test system was set up to perform performance measurements of the prototype system. A test lab, consisting of three dedicated workstations, was used to run the test applications. The workstations were connected using a non-dedicated 10-Mbps coaxial Ethernet network. The network layout of the test lab is shown in Figure 30. All workstations used (test01, test02 and test03) had a single Intel P-II 450 MHz processor with 512K of cache and 64MB RAM. All three workstations had only an MQOSD and an MRTSD running per workstation. In addition to the test workstations, the network contained two servers and many other workstations that were in constant use by other party. The traffic generated on the network from the uncontrolled workstations and servers was unknown and variable with time. When a dedicated network was required, the network was used off working hours. This provided a near-dedicated network where the traffic generated by the uncontrolled workstations was minimal.

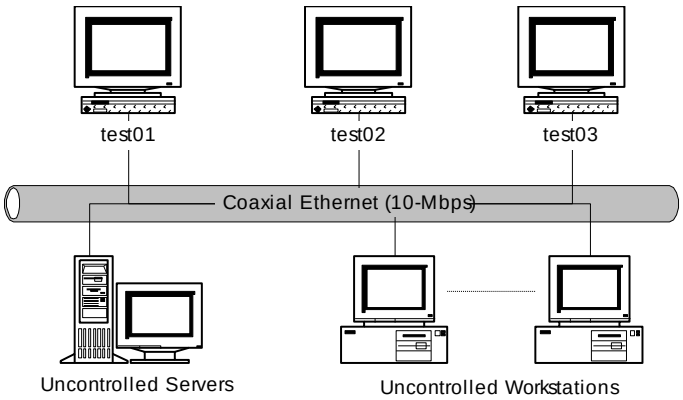


Figure 30 - Testing Platform

The tests used concentrated on measuring two aspects of the system: enhancements and costs. The experiments measure the enhancements made to the timely arrival of transmitted data. The experiments also measure the overheads incurred by the system to provide the measured enhancements.

5.1 Goodput Enhancement

Goodput is an invented term that denotes the percentage of transmitted packets that arrived to the receiver before or at their deadline time. The higher the goodput, the more comprehendible a transmitted multimedia flow becomes. A goodput of 100% means that all the packets in the flow were received on time by the receiver.

To measure goodput, three flows were created on the three workstations. Each sender sent 10,000 packets of 5-Kbytes each before terminating the flow. The pattern of sending the packets was dependent on the throughput being measured. The receiver timed the packets as they were received and discarded any packets that did not arrive on time. The ratio of the number of packets received on time to the total number of packets sent is the goodput of the flow. The total goodput of the network was the average goodput for the three flows. The same test was repeated with different throughputs. Varying the throughput allowed testing the effect of increasing the requested throughput on the goodput measured at the receiver. The test was further repeated using UDP/IP for sending the packets instead of the controlled QoS system. Figure 31 shows the measured goodput for different requested cumulative throughput. The cumulative throughput is the sum of the requested throughputs for all three flows.

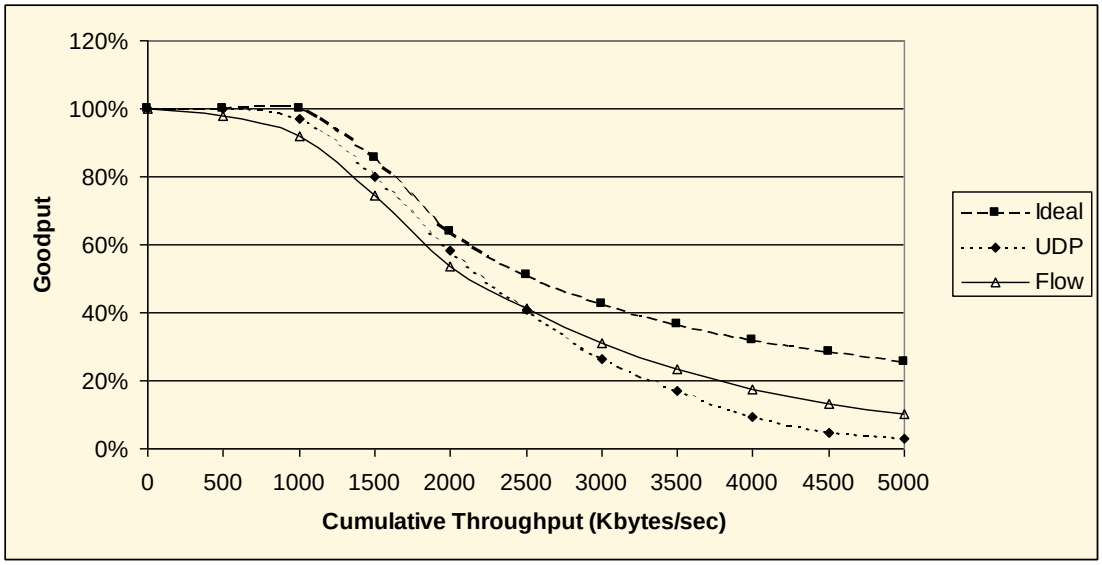


Figure 31 – Percentage of good packets at variable requested throughput

The ideal values for the goodput were calculated using the 10-Mbps (1280 Kbytes/sec) nominal bandwidth for the underlying network. The ideal network was assumed to only allow 1280 Kbytes to be transmitted per second. If more is requested, the ideal network will only pass 1280 Kbytes of the requested throughput and hence the percentage of correct packets would be estimated on this ground.

The QoS-controlled flows demonstrated a worse goodput at lower cumulative throughputs than that of normal UDP streams. This can be attributed to the overheads of QoS maintenance and monitoring which use a portion of the network bandwidth that could have otherwise been used by flow data. The advantages of the QoS system are much less than the overheads of the system at this stage. At higher requested throughputs, the controlled flows exhibit a higher goodput than normal UDP streams that send data using the same throughput. At this stage, the network bandwidth saved due to the realtime packet scheduling performed at the MRTSD is more than that

wasted by QoS maintenance and monitoring. At 5000 Kbytes/sec (approximately 1.6 Mbytes/sec per flow), 10% of the packets arrive on time when sent using the QoS system as opposed to only 3% arriving on time using UDP streams.

5.2 Throughput Overhead

The proposed QoS system was developed as a layer above UDP/IP. This ports all the overheads of UDP to the system and adds to them the overhead of the system itself. UDP has overheads due to data copying and checksum creation and validation. The proposed system adds throughput overheads due to packet scheduling.

One flow was created to send packets at the maximum rate possible. The flow attempts to send 10,000 packets of data in a tight loop. The experiment was repeated for different packet sizes and the throughput was measured at the receiver. The exercise was performed using QoS flows, UDP/IP and raw IP. Figure 32 shows the measured system throughput at different packet sizes.

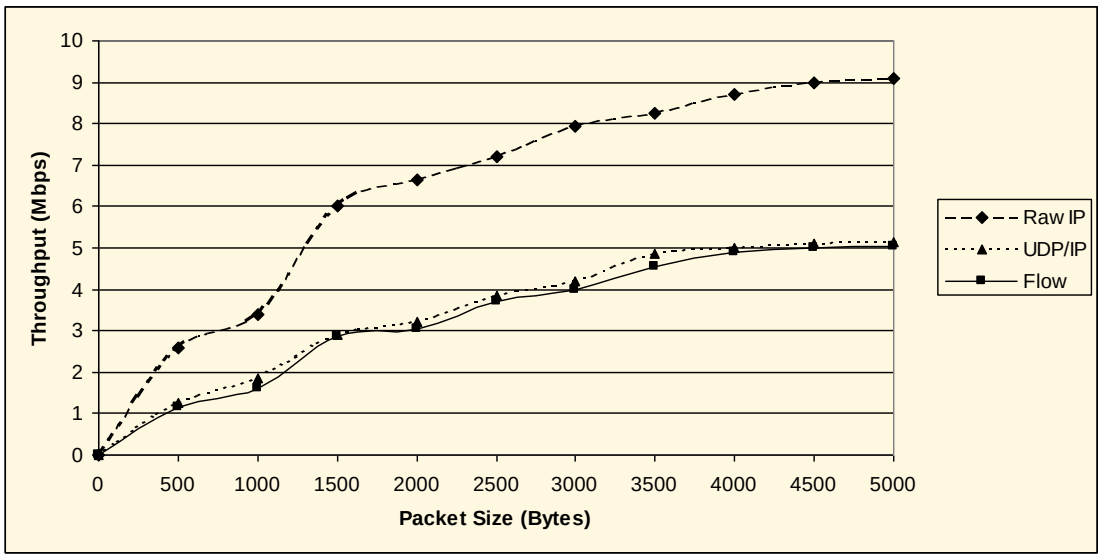


Figure 32 - Measured throughput for different packet sizes

Raw IP, expectedly, outperformed all higher level protocols. Raw IP is the basis of all other protocols and contains minimal overheads. UDP/IP performed at almost 50% of raw IP due to data-copying overhead as well as checksum generation and transport-level overheads. The throughput for the proposed system was slightly less than UDP/IP due to the overhead of the realtime packet scheduler.

5.3 Network Bandwidth

The network capacity used by the QoS system is not entirely used for the transmission of data packets. The QoS system uses probe packets for performing QoS monitoring. The percentage of network bandwidth used for QoS monitoring is determined according to a system-level parameter as described in section 4.2.6.

A single flow was created to send data at different QoS levels. Counters were inserted at the MRTSD to measure the total number of bytes sent for data packets and the corresponding number of bytes sent for probe packets. The system-level monitoring parameter was set at 0.1 (10%). The distribution of the utilized network bandwidth is shown in Figure 33.

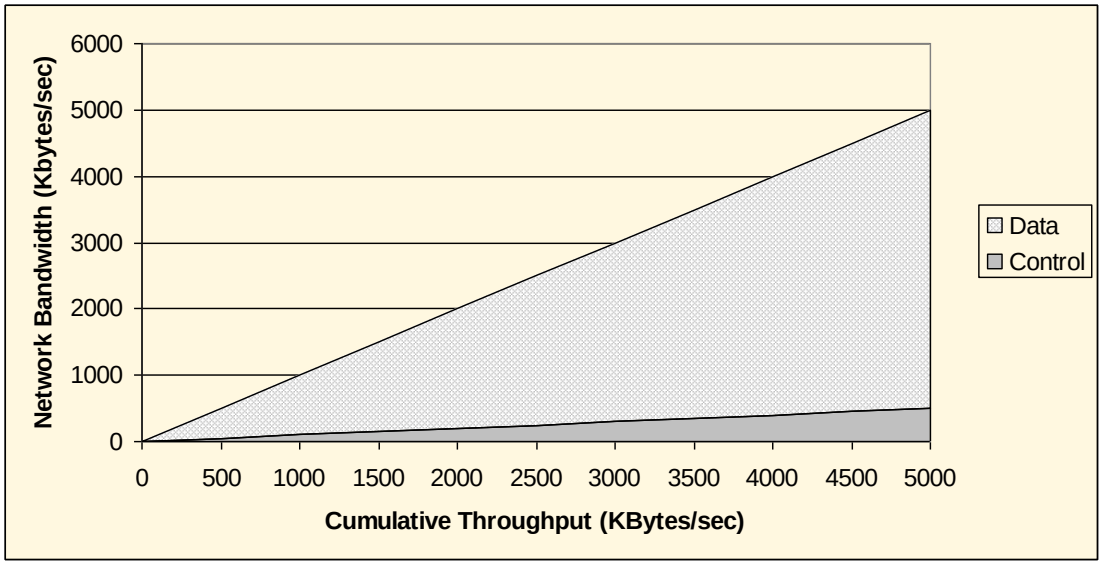


Figure 33 - Breakdown of network bandwidth utilization

The network bandwidth utilization was split into exactly 90% for data and 10% for probe packets. This is in agreement with the system-level monitoring parameter that was set at 0.1.

5.4 Delay Overhead

The added system layer in the proposed system adds to the latency of the overall network. First, there is the copying required by the sender to make the data ready for transmission. The data pointers to the shared memory are then transmitted using IPC to the MRTSD without actually copying the data. The MRTSD uses realtime scheduling to prioritize packet transmission, which adds to the delay of sending every packet. Figure 34 shows the breakdown of system latency into its different components.

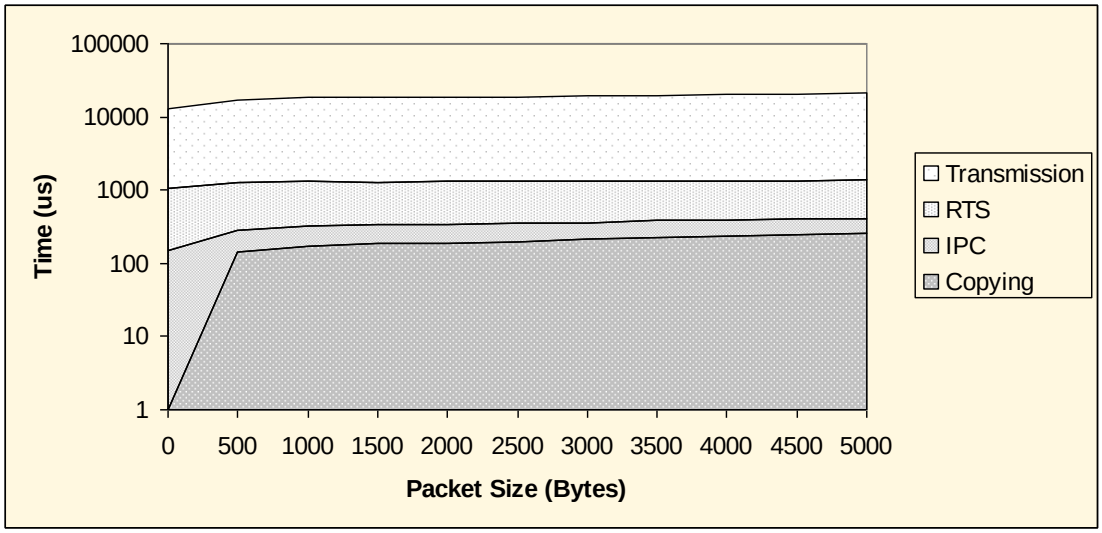


Figure 34 – System latency as a function of packet size *(using log-scale)*

The largest part of latency is that caused by the network itself. This includes the delay due to UDP, the network delay and the transmission time. The elements of the latency introduced by the QoS system are an average delay of approximately 1000μs for the realtime scheduler and a constant delay (approximately 150μs) for IPC transmission of the data pointers. There is also a variable delay for copying data into shared memory by the sender, which depends on the size of the data being copied. The dominant part of the QoS system delay is due to the realtime scheduler. This is attributed to the usage of semaphores for signaling the arrival and consumption of packets in the packet queue.

5.5 Summary

The measured performance of the proposed system indicates that realtime scheduling indeed helps in the correct multiplexing of multimedia data. Realtime scheduling avoids sending data that has expired and makes room for data that has a chance of reaching its destination on time. The added overhead in throughput is significant with

comparison to raw IP transmission. A system that is built directly on top of raw IP instead of UDP/IP would provide a higher throughput for applications. The added latency to the system (approx. 1300μs) is acceptable for high-latency networks such as the one used in the test system. This latency might not be acceptable in networks with lower-latency.

The experiments performed in this section did not count the effect of cooperation among multimedia flows. In reality, QoS notification significantly adds to the collective performance of multimedia applications by allowing applications to reduce their QoS requirements during network congestion and relaxing them again when the network is less congested. This can significantly increase goodput to allow more multimedia flows to coexist on the same network. Moreover, the effect of admission control was not tested. Admission control notably protects the network bandwidth from being overwhelmed by a flow whose requested QoS-level could not be satisfied.

Chapter 6 SUMMARY AND CONCLUSION

This thesis researched the different aspects of providing a QoS management solution for multimedia communication. QoS management functions were normally provided with networks that have an inherent capacity for reserving resources. The thesis aimed at adapting the concepts of QoS management to the world of non-guaranteed networks. This research has been conducted in the context of an environment where network performance cannot be guaranteed and where performance may be changing with time.

6.1 Development

An abstraction layer was developed on top of a standard transport protocol to provide QoS functionality. The abstraction layer used the concept of a flow, which is a stream of multimedia data that has specific quality constraints. The abstraction layer includes a QoS-management subsystem, a realtime scheduling subsystem, and a user-level library.

The QoS management subsystem (the MQOSD) provides a central point of control for QoS functions on every workstation. The MQOSD is the single point for active storage of flow-related QoS data. It performs QoS specification, mapping, and admission prior to flow creation by utilizing the stored QoS information in taking QoS decisions. The MQOSD also performs QoS notification for flows whose desired QoS levels could not be satisfied due to network constraints.

The realtime scheduling subsystem (the MRTSD) performs QoS maintenance functions on every workstation. It communicates with senders of the same workstation to receive and prioritize flow data packets before sending them to their destinations. The MRTSD is also responsible for QoS monitoring by measuring actual QoS values and reporting them to the MQOSD.

The user-level library provides a high-end interface to the users of the QoS system (senders). The library defines a highly portable interface on top of high-level languages. It allows senders to create flows, manage their QoS requirements as well as receive and handle notifications for QoS degradation. Receivers of multimedia flows are oblivious to whether the information they receive is QoS managed. Receivers are not involved in the QoS management process.

The complete system allows senders to create flows and simply specify their QoS requirements. It also manages the data being transferred to maximize the utilization of the network using the knowledge it has from the specified QoS requirements. This is performed while coexisting with other time-varying, non-controlled data on the same network.

6.2 Results

The core design features of the system were implemented on Linux using UDP/IP as the base transport protocol. User libraries were provided for C++ to extend QoS functionality to the language. IPC was used for local communication between the user libraries and the QoS subsystems (the MQOSD and the MRTSD).

The testing of the implemented system indicated that the system provides better network traffic management but with a high overhead. Goodput was used as a basis for measuring the performance of QoS management system. It indicates the percentage of flow data packets that arrived on time according to the contracted QoS levels for the flow. Goodput during network overload for the QoS managed system was significantly higher than an uncontrolled system. The system provided triple the goodput (10% versus 3%) when the total concurrent requests were triple the network capacity.

The runtime of the system encompassed a high throughput overhead. Being implemented on top of UDP/IP, the system provided a performance that was slightly lower than that of UDP/IP but almost half that of raw IP. At low throughput requests, the system provided performance that was poorer than that of uncontrolled systems due to the high system overhead. A system implemented on top of raw IP instead of UDP/IP would provide higher goodput results.

The overhead in overall system delay was not as severe as the throughput overhead. The end-to-end delay was increased by less than 10% of the minimum measured network delay during system testing. This is attributed to the usage of out-of-bound control channels. Sending control data on separate channels rather than using piggybacking mechanisms reduces the overhead of handling flow data and reduces the operations performed at the sender and receiver. This significantly reduced the increase in system delay. The notable constituent (90%) of the added delay is the delay caused by realtime scheduling of packets at the MRTSD. The algorithm used was an non-optimized Earliest Deadline First scheme.

6.3 Conclusions

The goal of this thesis has been to provide QoS management for flows on non-guaranteed networks. This goal has been achieved through the design of the model presented in this thesis. Instead of relying on the network to perform resource reservation for the specified QoS levels, the system actively performs QoS assessment and monitoring. The model also presents a novel predictive approach for QoS admission by anticipating current network QoS levels and intelligently admitting or denying acceptance of future QoS requests. QoS contracts are constantly monitored and evaluated to provide proper notification of any instances where the contracted levels are not met.

The proposed model provides a threshold level for QoS. Threshold QoS provides a higher level than the common best-effort QoS that is adopted in non-guaranteed networks. The testing and measurement of the proposed model has shown improvements in quality over best-effort QoS. This is to be compared to the compulsory QoS levels that are warranted in guaranteed networks. The adopted model provides a medium level between best-effort QoS and compulsory QoS.

Finally, the proposed model was implemented on a common LAN and showed that QoS managed flows can coexist with non-managed data streams. Applications were written using the high-level interface provided by the user libraries of the proposed system. Applications were able to work with the notion of threshold QoS and were able to coordinate at times of congestion to work with lower QoS levels and provide maximum quality for the available network resources.

6.4 Future Research

The proposed model provides a framework for the study of further aspects of QoS management. This work has mainly focused on providing breadth coverage of QoS functions to verify the feasibility of the concept. One QoS aspect, predictive QoS admission, was studied in depth. However, the proposed model provides a rich research platform for the investigation of the following aspects:

- Realtime scheduling
- Elaborate notification and adaptation to QoS feedback
- Comprehensive specification and mapping for QoS levels
- Multicast and group management
- QoS negotiation and renegotiation mechanisms

The proposed model implements a simple Earliest Deadline First realtime scheduling algorithm. The realtime-scheduling algorithm is currently a bottleneck in the proposed system. Effective realtime scheduling can increase network bandwidth utilization and provide better management for network traffic. This should lead to better system performance and increased goodput at higher requested QoS levels.

At present, the proposed model employs primitive QoS notification. Elaborate QoS notification should allow user specification of the events requiring notification. Moreover, it the mechanism should allow for intelligent distribution of the exceeded network capacity. Currently, the system notifies all active flows of QoS degradation in order to cooperate in reducing the requested QoS levels. This proves efficient to the network but too downgrading to the applications. Approaches that are more formal

include the identification of which flows contribute to the network congestion and only notifying those flows that are creating the problem.

Comprehensive QoS specification can allow a versatile set of parameters for specifying user-level QoS parameters. QoS specification should allow specifying all multimedia formats as well as having a generic way of defining the transmission patterns of flow data. Elaborate specification should also differentiate between stored data and data generated using live multimedia sources. This should allow QoS mapping to tighter network-level QoS parameters leading to effective management of network bandwidth.

The proposed work has been designed with the assumption of having only unicast flows. Typical multimedia applications involve multicast operation and group communication. The proposed system provides a basic framework that can be modified to include group handling functionality to allow group creation and membership. This is the first step in providing a complete multicast solution for multi-point communication. Moreover, multicast flows face the challenge of having different QoS levels requested by different receivers, which introduces the concept of multi QoS levels per flow.

The current model allows only acceptance or denial of flow creation. A more complete system should provide mechanisms for negotiating QoS levels. QoS renegotiation during operation is also essential to adequately adapt to the varying network performance. QoS negotiation and renegotiation mechanisms allow the system to recommend QoS levels to the applications instead of denying them the

service. It also allows applications to reach better agreement with the system on what QoS levels it requires.

Finally, it is hoped that continuous analysis and further development of the proposed system will lead to a more complete study of all aspects of QoS management on non-guaranteed networks.

REFERENCES

[Aurrecoechea 98] C. Aurrecoechea, A. Campbell and L. Hauw. “A Survey of QoS Architectures.” Multimedia Systems Journal, May 1998.

[Banerjea 97] A. Banerjea and H. Vin (ed.). “Heterogeneous Networking.” IEEE Multimedia, April-June 1997, pp. 84-87.

[Besse 94] L. Besse, et al. “Towards an Architecture for Distributed Multimedia Application Support.” Proceedings of International Conference on Multimedia Computing and Systems, Boston, May 1994.

[Blair 93] G. Blair, et al. “Summary of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV’93).” ACM Operating Systems Review, November 1993.

[Booch 94] Booch, Grady. “Object-Oriented Analysis and Design with Applications.” Second Edition. California: Benjamin-Cummings Co., 1994.

[Bosco 96] P. Bosco, et al. “The ReTINA Project: An Overview.” ReTINA Techinical Report 96-15, May 1996.

[Braden 94] R. Braden, D. Clark and S. Shenker. “Integrated Services in the Internet Architecture: An Overview.” Internet Network Working Group, RFC 1633, July 1994.

[Braden 96] R. Braden, et al. “Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specifications.” Internet Draft, October 1996.

[Braun 97] T. Braun and P. Liu (ed.). “Internet Protocols for Multimedia Communications.” IEEE Multimedia, July-September 1997, pp. 85-90.

[Campbell 93] A. Campbell, G. Coulson and F. Garcia. “Integrated Quality of Service for Multimedia Communications.” Proc. IEEE INFOCOM’93, San Francisco, USA, April 1993.

[Campbell 94] A. Campbell, G. Coulson and D. Hutchison. “A Quality of Service Architecture.” ACM SIGCOMM Computer Communication Review, April 1994.

[Campbell 96] A. Campbell, C. Aurrecoechea and L. Hauw. “A Review of QoS Architectures.” Proc. 4th IFIP International Workshop on QoS, Paris, March 1996, (invited paper).

[Campbell 97] A. Campbell and G. Coulson. “QoS Adaptive Transports: Delivering Scalable Media to the Desktop.” IEEE Network, March-April 1997, pp. 18-27.

[Deering 95] S. Deering and R. Hinden. “Internet Protocol Version 6 (IPv6) Specifications.” IETF RFC 1883, December 1995.

[Delgrossi 95] L. Delgrossi and L. Berger. “Internet Stream Protocol Version 2 (ST2): Protocol Specification – Version ST2+.” Internet Network Working Group, RFC 1819, August 1995.

[DeMeer 95] J. deMeer and A. Hafid. “QoS Modelling of Distributed Teleoperating Services.” ISN 1995.

[Ferrari 96] D. Ferrari. “The Tenet Experience and the Design of Protocols for Integrated Services Internetworks.” Multimedia Systems Journal, November 1995.

[Gecsie 97] J. Gecsei. “Adaptation in Distributed Multimedia Systems.” IEEE Mutlimedia, April-June 1997, pp. 58-66.

[Gopalakrishna 94] G. Gopalakrishna and G. Parulkar. “Efficient Quality of Service in Multimedia Computer Operating Systems.” Dept. of Computer Science, Washington University, Report WUCS-TM-9404, August 1994.

[Hafid 96a] A. Hafid. “Quality of Service Negotiation for Distributed Multimedia Applications.” *http://www.iro.umontreal.ca/labs/teleinfo/PubListIndex.html*.

[Hafid 96b] A. Hafid, G. Bochmann and R. Dssouli. “Distributed Multimedia Applications and Quality of Service.” *http://www.iro.umontreal.ca/labs/teleinfo/PubListIndex.html*. Technical report #1036. May 1996.

[Hutchison 95] D. Hutchison, et al. Network and Distributed Systems Management. M. Sloman (ed.). "Quality of Service Management in Distributed Systems." Addison Wesley, May 1995.

[IEEE 95] "Interface Requirements for Realtime Distributed Systems Communication." IEEE, July 1995.

[ISO 93a] "Information Technology – Digital Compression and Coding of Continuous-Tone Still Images." ISO/IEC/JTC 1/ IS 10918, 1993.

[ISO 93b] "Information Technology – Coding of Moving Pictures and Associated Audio for Digital Storage Media up to About 1.5 MBit/s" ISO/IEC/JTC 1/IS 11172, 1993.

[ISO 94] "Information Technology – Generic Coding of Moving Pictures and Associated Audio Information." ISO/IEC/JTC 1/DIS 13818, 1994.

[ISO 95] "Open Systems Interconnection, Data Management and Open Distributed Processing : QoS – Basic Framework." ISO/IEC JTC 1/SC 21 N9309, January 1995.

[Kerherve 94] B. Kerherve, et al. "On Distributed Multimedia Presentational Applications: Functional and Computational Architecture and QoS Negotiation." Proc. of High Speed Networks Conference, pp. 1-19, 1994.

[Kong 97] J. Kong and J. Hong. “A CORBA-based Management Framework for Distributed Multimedia Services and Applications.” DSOM 1997.

[Lazar 94] A. Lazar, S. Bhonsle and K. Lim. “A Binding Architecture for Multimedia Networks.” Proceedings of COST-237 Conference on Multimedia Transport and Teleservices, Vienna, Austria, 1994.

[Mitchell 97] S. Mitchell, et al. “The Djinn Framework for Distributed Multimedia Applications.”

[Nahrstedt 95] K. Nahrstedt. “An Architecture for End-to-End Quality of Service Provision and its Experimental Validation.” Ph.D. Thesis, University of Pennsylvania, 1995.

[OMG 96] “Realtime CORBA: A White Paper.” Object Management Group, December 1996.

[Reynolds 96] Reynolds, Peter. “End-to-End Quality of Service Support for Integrated Multimedia Workstations.” Masters Thesis. University of Wollongong. July 1996.

[Schulzrinne 95] H. Schulzrinne and S. Casner. “RTP: A Transport Protocol for Real-Time Applications.” Internet Draft, 1995.

[Siqueria 97] F. Siqueria. “A Framework for Distributed Multimedia Applications based on CORBA and Integrated Services Networks.

Ph.D. Project. Distributed Systems Group, Trinity College, Dublin.

[Stuttgen 95] H. Stüttgen. "Network Evolution and Multimedia Communication." IEEE Multimedia, Fall 1995, pp. 42-59.

[Vogel 95] A. Vogel, et al. "On QoS Negotiation in Distributed Multimedia Applications." *http://www.iro.umontreal.ca/labs/teleinfo/PubListIndex.html*. Technical report #977. May 1995.

[Volg 96] C. Volg, et al. "HeiRAT – Quality of Service Management for Distributed Multimedia Systems." Multimedia Systems Journal, November 1995.

[Worsley 97] J. Worsley and T. Ogunfunmi. "Isochronous Ethernet – An ATM Bridge for Multimedia Networking." IEEE Multimedia, January-March 1997, pp. 58-67.

[Yau 96a] D. Yau and S. Lam. "An Architecture Towards Efficient OS Support for Distributed Multimedia." Proc. IS&T/SPIE Multimedia Computing and Networking, pp. 424-435, San Jose, CA, January 1996.

[Yau 96b] D. Yau. "Operating System Support for Distributed Multimedia." Dissertation Proposal. Ph.D. Thesis. The University of Texas at Austin, June 1996.

[Yau 97] D. Yau and S. Lam. “Migrating Sockets for Networking with Quality of Service Guarantees.” ACM SIGCOMM, 1997.

[Yeadon 96] N. Yeadon, et al. “Filters: QoS Support Mechanisms for Multipeer Communications.” IEEE Journal on Selected Areas in Communication (JSAC), Issue on Distributed Multimedia Systems and Technology, 3rd Quarter 1996.

[Yun 97] T. Yun, J. Kong and J. Hong. “Object-Oriented Modeling of Distributed Multimedia Services.” ICCS, 1997.

Appendix A: Neural Network Training

The neural network used as the second step of QoS admission was trained using MATLAB. A feed-forward neural network was defined using MATLAB to match the network shown in Figure 21. The network was trained using the back-propagation algorithm using the following data:

Throughput Acceptability	Delay Acceptability	Error-Rate Acceptability	Decision [1=accept, 0=deny]
0.00	0.00	0.00	0
0.25	0.00	0.00	0
0.50	0.00	0.00	0
0.75	0.00	0.00	0
1.00	0.00	0.00	0
0.00	0.25	0.25	0
0.25	0.25	0.25	0
0.50	0.25	0.25	0
0.75	0.25	0.25	0
1.00	0.25	0.25	0
0.00	0.50	0.50	0
0.25	0.50	0.50	0
0.50	0.50	0.50	0
0.75	0.50	0.50	0
1.00	0.50	0.50	1
0.00	0.75	0.75	0
0.25	0.75	0.75	0
0.50	0.75	0.75	0
0.75	0.75	0.75	1
1.00	0.75	0.75	1
0.00	1.00	1.00	0
0.25	1.00	1.00	0
0.50	1.00	1.00	1
0.75	1.00	1.00	1
1.00	1.00	1.00	1
0.00	0.25	0.00	0
0.00	0.50	0.00	0
0.00	0.75	0.00	0
0.00	1.00	0.00	0
0.25	0.00	0.25	0
0.25	0.50	0.25	0
0.25	0.75	0.25	0
0.25	1.00	0.25	0

Throughput Acceptability	Delay Acceptability	Error-Rate Acceptability	Decision [1=accept, 0=deny]
0.50	0.00	0.50	0
0.50	0.25	0.50	0
0.50	0.75	0.50	0
0.50	1.00	0.50	0
0.75	0.00	0.75	0
0.75	0.25	0.75	0
0.75	0.50	0.75	1
0.75	1.00	0.75	1
1.00	0.00	1.00	0
1.00	0.25	1.00	0
1.00	0.5	1.00	1
1.00	0.75	1.00	1
0.00	0.00	0.25	0
0.00	0.00	0.50	0
0.00	0.00	0.75	0
0.00	0.00	1.00	0
0.25	0.25	0.00	0
0.25	0.25	0.50	0
0.25	0.25	0.75	0
0.25	0.25	1.00	0
0.50	0.50	0.00	0
0.50	0.50	0.25	0
0.50	0.50	0.75	0
0.50	0.50	1.00	1
0.75	0.75	0.00	0
0.75	0.75	0.25	0
0.75	0.75	0.50	1
0.75	0.75	1.00	1
1.00	1.00	0.00	0
1.00	1.00	0.25	0
1.00	1.00	0.50	1
1.00	1.00	0.75	1

The training set was developed using common sense. The strategy was to accept flows when the inputs were high and to reject flows that had low inputs. Preference was given to throughput as the most important input. A flow needs to have high throughput acceptability but only medium delay and error rate acceptability to get accepted into the network.

The set of weights and biases obtained after training the network were stored in a text file called “nn-weights.dat” for usage by the MQOSD during runtime.

The following file was used for creating and training the neural network in MATLAB version 4.1:

```
% INITIALIZE
inputrange = [0 1; 0 1; 0 1];
[w1, b1, w2, b2] = initff (inputrange, 3, 'logsig', 1,
'logsig');

% TRAIN
trainingset = [
0.00  0.00  0.00  0;
0.25  0.00  0.00  0;
0.50  0.00  0.00  0;
0.75  0.00  0.00  0;
1.00  0.00  0.00  0;

0.00  0.25  0.25  0;
0.25  0.25  0.25  0;
0.50  0.25  0.25  0;
0.75  0.25  0.25  0;
1.00  0.25  0.25  0;

0.00  0.50  0.50  0;
0.25  0.50  0.50  0;
0.50  0.50  0.50  0;
0.75  0.50  0.50  0;
1.00  0.50  0.50  1;

0.00  0.75  0.75  0;
0.25  0.75  0.75  0;
0.50  0.75  0.75  0;
0.75  0.75  0.75  1;
1.00  0.75  0.75  1;

0.00  1.00  1.00  0;
0.25  1.00  1.00  0;
0.50  1.00  1.00  1;
0.75  1.00  1.00  1;
1.00  1.00  1.00  1;

0.00  0.25  0.00  0;
0.00  0.50  0.00  0;
0.00  0.75  0.00  0;
0.00  1.00  0.00  0;

0.25  0.00  0.25  0;
0.25  0.50  0.25  0;
0.25  0.75  0.25  0;
0.25  1.00  0.25  0;

0.50  0.00  0.50  0;
0.50  0.25  0.50  0;
0.50  0.75  0.50  0;
0.50  1.00  0.50  0;

0.75  0.00  0.75  0;
```

```
0.75  0.25  0.75  0;
0.75  0.50  0.75  1;
0.75  1.00  0.75  1;

1.00  0.00  1.00  0;
1.00  0.25  1.00  0;
1.00  0.5   1.00  1;
1.00  0.75  1.00  1;

0.00  0.00  0.25  0;
0.00  0.00  0.50  0;
0.00  0.00  0.75  0;
0.00  0.00  1.00  0;

0.25  0.25  0.00  0;
0.25  0.25  0.50  0;
0.25  0.25  0.75  0;
0.25  0.25  1.00  0;

0.50  0.50  0.00  0;
0.50  0.50  0.25  0;
0.50  0.50  0.75  0;
0.50  0.50  1.00  1;

0.75  0.75  0.00  0;
0.75  0.75  0.25  0;
0.75  0.75  0.50  1;
0.75  0.75  1.00  1;

1.00  1.00  0.00  0;
1.00  1.00  0.25  0;
1.00  1.00  0.50  1;
1.00  1.00  0.75  1;
]';

p=trainingset(1:3,1:size(trainingset,2));
t=trainingset(4,1:size(trainingset,2));
tp=[100 3000 nan nan nan nan nan nan];

[w1, b1, w2, b2, te, tr] = trainbpx (w1, b1, 'logsig', w2, b2,
'logsig', p, t, tp);

% SIMULATE
i=[1;1;1];
while i(1)~=0 | i(2)~=0 | i(3)~=0,
   i = input('input? ')';

x1 = (w1*i);
x2 = x1+b1*ones(1,size(x1,2));
x3 = 1 ./ (1+exp(-x2));

x4 = (w2*x3);
x5 = x4+b2*ones(1,size(x4,2));
x6 = 1 ./ (1+exp(-x5));
r = x6

   simuff (i, w1, b1, 'logsig', w2, b2, 'logsig')
end
```

Appendix B: Object-Oriented Design Class Details

QoS Structures

```
#ifndef QOS_H
#define QOS_H

#include <iostream.h>

struct NetworkQoS {
   int throughput;
   int delay;
   int jitter;
   int errorrate;
   NetworkQoS &operator =(const NetworkQoS &src);
   int operator <(const NetworkQoS &rhs);
   friend ostream &operator <<(ostream &ostream, NetworkQoS
&qos);
};

struct AudioQoS {
   int freq;
   int channels;
   int samplesize;
};

struct VideoQoS {
   struct {
      int x, y;
   } resolution;
   int colors;
   int frames;
};

struct UserQoS {
   enum {
      AUDIO, VIDEO
   } datatype;
   union {
      struct AudioQoS audio;
      struct VideoQoS video;
   } data;
   int compression;
   enum {
      INTERACTIVE, NONINTERACTIVE
   } interactivity;
   int tolerance;

   UserQoS &operator =(const UserQoS &src);
   friend ostream &operator <<(ostream &ostream, UserQoS &qos);
};
#endif
```

Buffer Class

```
#ifndef BUFFER_H
#define BUFFER_H

// if not in Linux, inline may not work (undefine it), and NULL
might
// not be defined (include stdio.h to define it)
#ifndef LINUX
#define inline
#include <stdio.h>
#endif

#include <iostream.h>

class Buffer
{
 protected:
   struct {
      int size;                       // allocated size for
buffer
      int extra;                      // extra bytes allocated
at beginning
                                      //   of buffer for
application use
      int localbuffer;                // buffer allocated here
or by caller
      char *buffer;                   // buffer pointer
      int inptr, outptr;              // indeces to current
positions in buffer
      int opaquedatasize;             // size of opaque data to
be sent
                                      //   opaque data (void*)
when being
                                      //   marshalled take the
first int
                                      //   before them as the
size. that
                                      //   int gets sent before
the data.
   } data;
   char *bufferdata;

 public:
   Buffer (unsigned short bufsize=8196, unsigned short
extrasize=0,
         char *buf=NULL);
   Buffer (Buffer &rhs);
   virtual ~Buffer ();

   inline void clear ();
   inline void rewind ();
   inline int maxsize ();
   inline int datasize ();
   inline int extrasize ();

   inline char *databuffer ();
   inline char *wholebuffer ();
   void setdatasize (int datasize);
   void setbuffer (char *buffer);

   void extra (void *extrabuf, int bufsize);

   virtual Buffer &operator =(Buffer &rhs);
```

```
   virtual Buffer &operator <<(Buffer &rhs);

   virtual Buffer &operator <<(int n);
   virtual Buffer &operator <<(short n);
   virtual Buffer &operator <<(long n);
   virtual Buffer &operator <<(unsigned n);
   virtual Buffer &operator <<(unsigned short n);
   virtual Buffer &operator <<(unsigned long n);
   virtual Buffer &operator <<(char c);
   virtual Buffer &operator <<(char *s);
   virtual Buffer &operator <<(void *v);

   virtual Buffer &operator >>(int &n);
   virtual Buffer &operator >>(short &n);
   virtual Buffer &operator >>(long &n);
   virtual Buffer &operator >>(unsigned &n);
   virtual Buffer &operator >>(unsigned short &n);
   virtual Buffer &operator >>(unsigned long &n);
   virtual Buffer &operator >>(char &c);
   virtual Buffer &operator >>(char *s);
   virtual Buffer &operator >>(void *v);
};

#endif
```

Comm Class

```
#ifndef COMM_H
#define COMM_H

#include "buffer.h"

class Comm
{
 public:
   virtual int Send (Buffer &buffer)=0;
   virtual int Receive (Buffer &buffer)=0;
   virtual int NonBlockingReceive (Buffer &buffer)=0;
};

#endif
```

ControlPacket Class

```
#ifndef CONTROLPACKET_H
#define CONTROLPACKET_H

#include "buffer.h"

class ControlPacket : public Buffer
{
 public:
   enum PacketType {
      KILL,
      // MQOSD packets
      CREATE, CREATEREPLY, TERMINATE, SELECTQOS,
SELECTQOSREPLY, NOTIFY,
      PINGDELAY, STARTPINGTHROUGHPUT, PINGTHROUGHPUT,
      PINGTHROUGHPUTREPLY,
      // MRTSD packets
      SEND, QUERY, QUERYREPLY, REPORT
   };
 public:
   ControlPacket (unsigned short bufsize, unsigned short
extrasize, char *buf);
   virtual PacketType Type ();
   virtual void Type (PacketType type);
};

#endif
```

CountTable Class

```
#ifndef COUNTTABLE_H
#define COUNTTABLE_H

#include "flowtable.h"

class CountTable : public FlowTable {
 protected:
   struct Entry {
      int count;
      long packetmax;
      long packetmin;
      long packetavg;
   };
 protected:
   virtual void DumpEntry (int index);
   virtual void SetEntry (int index, void *data);
   virtual void DeleteEntry (int index);
 public:
   CountTable (int size);
   void Inc (flow_t flowid, long packetsize);
   void Reset (flow_t flowid);
   int Count (flow_t flowid);
   long AveragePacketSize (flow_t flowid);
};

#endif
```

CreatePacket Class

```
#ifndef CREATEPACKET_H
#define CREATEPACKET_H

#include "flow.h"
#include "qos.h"
#include "controlpacket.h"
#include <unistd.h>

#include "pid.h"

class CreatePacket : public ControlPacket
{
 protected:
   struct PacketCreate {
      PacketType type;
      pid_t pid;
      pid_t notifypid;
      UserQoS qos;
      char address[30];
      unsigned short port;
   };
   PacketCreate *packet;
 public:
   CreatePacket (unsigned short extrasize=0, char *buf=NULL);
   pid_t pid ();
   void notifypid (pid_t notifypid);
   pid_t notifypid ();
   void pid (pid_t pid);
   void qos (UserQoS *qos);
   void qos (UserQoS &qos);
   char *address ();
   void address (char *address);
   unsigned short port ();
   void port (unsigned short port);
};

#endif
```

CreateReplyPacket Class

```
#ifndef CREATEREPLYPACKET_H
#define CREATEREPLYPACKET_H

#include "flow.h"
#include "controlpacket.h"
#include <unistd.h>

class CreateReplyPacket : public ControlPacket
{
 protected:
   struct PacketCreateReply {
      PacketType type;
      flow_t flowid;
//      int qos;
   };
   PacketCreateReply *packet;
 public:
   CreateReplyPacket (unsigned short extrasize=0, char
*buf=NULL);
   flow_t flowid ();
   void flowid (flow_t flowid);
//   int qos ();
//   void qos (int qos);
};

#endif
```

Flow Class

```
#ifndef FLOW_H
#define FLOW_H

#include "ipc.h"

// if not in Linux, inline may not work (undefine it)
#ifndef LINUX
#define inline
#endif

#define FLOWERROR (0)

typedef unsigned int flow_t;

class Flow
{
 protected:
   IPC *mqosd;               // mqosd ipc channel
   IPC *mrtsd;               // mrtsd ipc channel
   flow_t flowid;

 public:
   Flow ();
   virtual ~Flow ();

   inline flow_t ID();
};

#endif
```

FlowTable Class

```
#ifndef FLOWTABLE_H
#define FLOWTABLE_H

#define FLOWTABLE_EMPTY (0)

// if not in Linux, inline may not work (undefine it)
#ifndef LINUX
#define inline
#endif

#include "flow.h"

class FlowTable {
 protected:
   struct Entry {
      flow_t flowid;
      void *data;
   } *table;
   int tablesize;
 protected:
   inline int Empty (int index);
   virtual int Find (flow_t flowid);
   virtual int AddEntry (flow_t flowid, void *data);
   // entry functions
   virtual void DeleteEntry (int index)=0;
   virtual void SetEntry (int index, void *data)=0;
   virtual void DumpEntry (int index)=0;
 public:
   FlowTable (int size);
   virtual ~FlowTable ();
   virtual int Delete (flow_t flowid);
   virtual void Dump ();
};

#endif
```

IPC Class

```
#ifndef IPC_H
#define IPC_H

// if not in Linux, inline may not work (undefine it)
#ifndef LINUX
#define inline
#endif

#include "comm.h"
#include "buffer.h"

class IPC : public Comm
{
 public:
   enum JoinType {
      CREATE, ATTACH
   };
 protected:
   JoinType join;
   int key;
   int msqid;
   long sendmsgtype;          // type of messages to send using
msgsnd()
   long receivemsgtype;       // type of messages to recv using
msgrcv()
 public:
   IPC (int remotekey, JoinType jointype=ATTACH);
   IPC (IPC &rhs);
   virtual ~IPC ();

   inline int MsgQueueID ();
   inline void SendType (long msgtype);
   inline void ReceiveType (long msgtype);

   virtual int Send (Buffer &buffer);
   virtual int Receive (Buffer &buffer);
   virtual int NonBlockingReceive (Buffer &buffer);
};

#endif
```

KillPacket Class

```
#ifndef KILLPACKET_H
#define KILLPACKET_H

#include "controlpacket.h"

class KillPacket : public ControlPacket
{
 protected:
   struct PacketKill {
      PacketType type;
   };
   PacketKill *packet;
 public:
   KillPacket (unsigned short extrasize=0, char *buf=NULL);
};

#endif
```

MQOSD Class

```
#ifndef MQOSD_H
#define MQOSD_H

#define MQOSD_IPCPORT (7776)
#define MQOSD_UDPPORT (7776)

#define MQOSD_IPCTOMQOSD (1)
#define MQOSD_IPCTOMRTSD (2)

#include "qos.h"
#include "qostable.h"
#include "pid.h"

#include <unistd.h>

//ASSUME: max concurrent flows = 50
#define MQOSD_MAXCONCURRENTFLOWS (50)
//ASSUME: max concurrent pinging flows = 50
#define MQOSD_MAXCONCURRENTPINGS (50)
//ASSUME: max ping request buffer size = 8K
#define MQOSD_MAXPINGBUFFER (8*1024)
//ASSUME: percentage of ping packets expected = 80%
#define MQOSD_PERCENTPINGPACKETS (80)
//ASSUME: max receive buffer size = 8K
#define MQOSD_MAXRECEIVEBUFFER (8*1024)
//ASSUME: ratio of old qos retained = 40% old, 60% new
#define MQOSD_OLDDATARATIO (40)

class MQoSDaemon
{
 protected:
   IPC *ipc;
   QoSTable *qostable;
 protected:
   // functions available to callers
   void Create (pid_t pid, pid_t notifypid, UserQoS &qos,
            char *address, unsigned short port);
   void Terminate (flow_t flowid);
   void SelectQoS (flow_t flowid, UserQoS &qos);
   void Query (flow_t flowid, long requestor);
   void Report (flow_t flowid, NetworkQoS &qos);

   // internal functions
   virtual int AdmitQoS (NetworkQoS &qos);
   virtual void MapQoS (UserQoS &userqos, NetworkQoS *netqos);
 public:
   MQoSDaemon (int maxflows);
   virtual ~MQoSDaemon ();
   static void *HandlePings (void *arg);
   static void *HandleRequests (void *arg);
};

#endif
```

MRTSD Class

```
#ifndef MRTSD_H
#define MRTSD_H

#include <semaphore.h>
#include <pthread.h>

#include "packettable.h"
#include "counttable.h"
#include "ipc.h"

#define MRTSD_IPCPORT (7777)
#define MRTSD_UDPPORT (7777)

#define MRTSD_IPCTOMRTSD (1)

//ASSUME: max mrtsd received packet size = 8K
#define MRTSD_MAXPACKETSIZE (8*1024)
//ASSUME: max packets awaiting send in mrtsd buffer = 500
#define MRTSD_MAXBUFFERPACKETS (100)
//ASSUME: max concurrent flows = 50
#define MRTSD_MAXCONCURRENTFLOWS (50)

#define TIMEDIFF(a,b) ((a.tv_sec-b.tv_sec)*1000000+(a.tv_usec-
b.tv_usec))
//TODO: make num of ping packets a user parameter
#define PING_NUMPACKETS (10)
//TODO: use alpha from user qos
#define PING_ALPHA (50)

class MRTSDaemon {
 protected:
   sem_t sema; // semaphore to synch receiver and scheduler
               // receiver posts when new packet is put in
buffer
               // scheduler waits until receiver posts a packet
   pthread_mutex_t mutex;
   PacketTable *packettable;
   CountTable *counttable;
   IPC *ipc;
   int currentping;
 protected:
   virtual void Ping (flow_t flowid, char *address);
 public:
   MRTSDaemon (int maxtables, int maxflows);
   virtual ~MRTSDaemon ();

   static void *Receiver (void *arg);
   static void *Scheduler (void *arg);
};

#endif
```

NotifyPacket Class

```
#ifndef NOTIFYPACKET_H
#define NOTIFYPACKET_H

#include "qos.h"
#include "controlpacket.h"

class NotifyPacket : public ControlPacket
{
 protected:
   struct PacketNotify {
      PacketType type;
      NetworkQoS actualqos;
      NetworkQoS desiredqos;
   };
   PacketNotify *packet;
 public:
   NotifyPacket (unsigned short extrasize=0, char *buf=NULL);
   void actualqos (NetworkQoS *qos);
   void actualqos (NetworkQoS &qos);
   void desiredqos (NetworkQoS *qos);
   void desiredqos (NetworkQoS &qos);
};

#endif
```

PacketTable Class

```
#ifndef PACKETTABLE_H
#define PACKETTABLE_H

#include "flowtable.h"
#include "qos.h"

#include <unistd.h>
#include <sys/time.h>

#include "pid.h"

class PacketTable : public FlowTable {
 protected:
   int numentries,maxnumentries,requests;    //TODO: remove
these two vars
   struct Entry {
      pid_t pid;
      char address[30];
      unsigned short port;
      struct timeval deadline;
      int shmkey;
      int datasize;
   };
 protected:
   virtual void DumpEntry (int index);
   virtual void SetEntry (int index, void *data);
   virtual void DeleteEntry (int index);
 public:
   PacketTable (int size);
   virtual ~PacketTable ();
   flow_t Add (flow_t flowid, pid_t pid, char *address,
unsigned short port,
              long sec, long usec, int shmkey, int datasize);
   int GetData (flow_t flowid, pid_t *pid, char *address,
unsigned short *port,
            long *sec, long *usec, int *shmkey, int *datasize);
   flow_t GetEarliestDeadline ();
};

#endif
```

PingDelayPacket Class

```
#ifndef PINGDELAYPACKET_H
#define PINGDELAYPACKET_H

#include "controlpacket.h"

class PingDelayPacket : public ControlPacket
{
 protected:
   struct PacketPingDelay {
      PacketType type;
   };
   PacketPingDelay *packet;
 public:
   PingDelayPacket (unsigned short extrasize=0, char
*buf=NULL);
};

#endif
```

PingThroughputPacket Class

```
#ifndef PINGTHROUGHPUTPACKET_H
#define PINGTHROUGHPUTPACKET_H

#include "flow.h"
#include "controlpacket.h"

#define PINGTHROUGHPUT_PACKETSIZE (5*1024)

class PingThroughputPacket : public ControlPacket
{
 protected:
   struct PacketPingThroughput {
      PacketType type;
      flow_t flowid;
      char data[PINGTHROUGHPUT_PACKETSIZE];
   };
   PacketPingThroughput *packet;
 public:
   PingThroughputPacket (unsigned short extrasize=0, char
*buf=NULL);
   flow_t flowid ();
   void flowid (flow_t flowid);
};

#endif
```

PingThroughputReplyPacket Class

```
#ifndef PINGTHROUGHPUTREPLYPACKET_H
#define PINGTHROUGHPUTREPLYPACKET_H

#include "controlpacket.h"

class PingThroughputReplyPacket : public ControlPacket
{
 protected:
   struct PacketPingThroughputReply {
      PacketType type;
      long timetaken;
   };
   PacketPingThroughputReply *packet;
 public:
   PingThroughputReplyPacket (unsigned short extrasize=0, char
*buf=NULL);
   void timetaken (long timetaken);
   long timetaken (void);
};

#endif
```

QoSAdmissionTable Class

```
#ifndef QOSADMISSIONTABLE_H
#define QOSADMISSIONTABLE_H

#include "qos.h"

class QoSAdmissionTable
{
 protected:
   enum FuzzyDigit { LOW, MIDLOW, MID, MIDHIGH, HIGH, NONE };
   struct FuzzyValue {
      FuzzyDigit fuzzy;
      int weight;
      FuzzyValue &operator =(FuzzyValue &rhs) {
       fuzzy = rhs.fuzzy;
       weight = rhs.weight;
       return *this;
      }
   };
   NetworkQoS networkstatus;
   NetworkQoS userrequest;
   FuzzyDigit decisiontable[5][5] =
     {   { HIGH,    HIGH,    HIGH,    HIGH,   HIGH   },
       { HIGH,    HIGH,    MIDHIGH, MID,    MID    },
       { HIGH,    MIDHIGH, MID,     MID,    MIDLOW },
       { MIDHIGH, MID,     MID,     MIDLOW, LOW    },
       { MID,     MIDLOW,  MIDLOW,  LOW,    LOW    }   };
 protected:
   void Fuzzify (long value, FuzzyValue *low, FuzzyValue
*high);
   int FuzzyDecide (FuzzyValue &requestlow, FuzzyValue
&requesthigh,
                FuzzyValue &networklow, FuzzyValue
&networkhigh);
   int NeuralDecide (int throughput, int delay, int errorrate);
   void FuzzyAnd (FuzzyValue &value1, FuzzyValue &value2,
FuzzyValue *result);
 public:
   void InputNetworkStatus (NetworkQoS &status);
   void InputUserRequest (NetworkQoS &request);
   int Decision ();
};

#endif
```

QoSTable Class

```
#ifndef QOSTABLE_H
#define QOSTABLE_H

#include "flowtable.h"
#include "qos.h"

#include <unistd.h>

#include "pid.h"

class QoSTable : public FlowTable {
 protected:
   flow_t nextflowid;
   struct Entry {
      pid_t pid;
      pid_t notifypid;
      NetworkQoS desiredqos;
      NetworkQoS actualqos;
      char address[30];
      unsigned short port;
   };
 protected:
   virtual void DumpEntry (int index);
   virtual void SetEntry (int index, void *data);
   virtual void DeleteEntry (int index);
 public:
   QoSTable (int size);
   flow_t Add (pid_t pid, pid_t notifypid, char *address,
unsigned short port, NetworkQoS &qos);
   int SetDesiredQoS (flow_t flowid, NetworkQoS &qos);
   int SetActualQoS (flow_t flowid, NetworkQoS &qos);
   int SetDestination (flow_t flowid, char *address, unsigned
short port);
   int GetDesiredQoS (flow_t flowid, NetworkQoS *qos);
   int GetActualQoS (flow_t flowid, NetworkQoS *qos);
   int GetData (flow_t flowid, pid_t *pid, pid_t *notifypid,
            NetworkQoS *desiredqos, NetworkQoS *actualqos,
            char *address, unsigned short *port);
   void GetQoSTotal (NetworkQoS *desiredqos, NetworkQoS
*actualqos);
   void GetQoSMax (NetworkQoS *desiredqos, NetworkQoS
*actualqos);
   //TODO: void GetQoSMin (NetworkQoS *desiredqos, NetworkQoS
*actualqos);
   void GetQoSAverage (NetworkQoS *desiredqos, NetworkQoS
*actualqos);
};

#endif
```

QueryPacket Class

```
#ifndef QUERYPACKET_H
#define QUERYPACKET_H

#include "flow.h"
#include "controlpacket.h"
#include <unistd.h>

class QueryPacket : public ControlPacket
{
 protected:
   struct PacketQuery {
      PacketType type;
      flow_t flowid;
      long requestor;
   };
   PacketQuery *packet;
 public:
   QueryPacket (unsigned short extrasize=0, char *buf=NULL);
   flow_t flowid ();
   void flowid (flow_t flowid);
   long requestor ();
   void requestor (long requestor);
};

#endif
```

QueryReplyPacket Class

```
#ifndef QUERYREPLYPACKET_H
#define QUERYREPLYPACKET_H

#include "qos.h"
#include "flow.h"
#include "controlpacket.h"
#include <unistd.h>

#include "pid.h"

class QueryReplyPacket : public ControlPacket
{
 protected:
   struct PacketQueryReply {
      PacketType type;
      pid_t pid;
      NetworkQoS actualqos;
      NetworkQoS desiredqos;
      char address[30];
      unsigned short port;
   };
   PacketQueryReply *packet;
 public:
   QueryReplyPacket (unsigned short extrasize=0, char
*buf=NULL);
   pid_t pid ();
   void pid (pid_t pid);
   void actualqos (NetworkQoS *qos);
   void actualqos (NetworkQoS &qos);
   void desiredqos (NetworkQoS *qos);
   void desiredqos (NetworkQoS &qos);
   char *address ();
   void address (char *address);
   unsigned short port ();
   void port (unsigned short port);
};

#endif
```

ReceiverFlow Class

```
#ifndef RECEIVERFLOW_H
#define RECEIVERFLOW_H

#include "flow.h"
#include "buffer.h"

class ReceiverFlow : public Flow
{
 protected:
 public:
 protected:
 public:
   int Create (Addr &addr, QoS &qos);
   int Terminate ();
   int SelectQoS (QoS &qos);
   int Receive (Buffer &buffer);
};

#endif
```

ReportPacket Class

```
#ifndef REPORTPACKET_H
#define REPORTPACKET_H

#include "qos.h"
#include "flow.h"
#include "controlpacket.h"

class ReportPacket : public ControlPacket
{
 protected:
   struct PacketReport {
      PacketType type;
      flow_t flowid;
      NetworkQoS qos;
   };
   PacketReport *packet;
 public:
   ReportPacket (unsigned short extrasize=0, char *buf=NULL);
   flow_t flowid ();
   void flowid (flow_t flowid);
   void qos (NetworkQoS *qos);
   void qos (NetworkQoS &qos);
};

#endif
```

SelectQoSPacket Class

```
#ifndef SELECTQOSPACKET_H
#define SELECTQOSPACKET_H

#include "qos.h"
#include "flow.h"
#include "controlpacket.h"
#include <unistd.h>

class SelectQoSPacket : public ControlPacket
{
 protected:
   struct PacketSelectQoS {
      PacketType type;
      flow_t flowid;
      UserQoS qos;
   };
   PacketSelectQoS *packet;
 public:
   SelectQoSPacket (unsigned short extrasize=0, char
*buf=NULL);
   flow_t flowid ();
   void flowid (flow_t flowid);
   void qos (UserQoS *qos);
   void qos (UserQoS &qos);
};

#endif
```

SelectQoSReplyPacket Class

```
#ifndef SELECTQOSREPLYPACKET_H
#define SELECTQOSREPLYPACKET_H

#include "flow.h"
#include "controlpacket.h"
#include <unistd.h>

class SelectQoSReplyPacket : public ControlPacket
{
 protected:
   struct PacketSelectQoSReply {
      PacketType type;
      flow_t flowid;
//      int qos;
   };
   PacketSelectQoSReply *packet;
 public:
   SelectQoSReplyPacket (unsigned short extrasize=0, char
*buf=NULL);
   flow_t flowid ();
   void flowid (flow_t flowid);
//   int qos ();
//   void qos (int qos);
};

#endif
```

SenderFlow Class

```
#ifndef SENDERFLOW_H
#define SENDERFLOW_H

#include "qos.h"
#include "flow.h"
#include "shmbuffer.h"

#include <pthread.h>
#include <semaphore.h>

class SenderFlow : public Flow
{
 protected:
   pid_t callerpid;
   pid_t threadpid;
   sem_t threadpidsem;
   pthread_t notificationhandler;
   virtual void QoSNotification (NetworkQoS &desiredqos,
NetworkQoS &actualqos)=0;
   static void *NotificationHandler (void *arg);
 public:
   SenderFlow ();
   virtual ~SenderFlow ();
   virtual flow_t Create (char *addr, unsigned short port,
UserQoS &qos);
   virtual void Terminate ();
   virtual int SelectQoS (UserQoS &qos);
   virtual int Send (SharedMemBuffer &buffer);
   virtual int Assess ();
};

#endif
```

SendPacket Class

```
#ifndef SENDPACKET_H
#define SENDPACKET_H

#include "flow.h"
#include "controlpacket.h"
#include <unistd.h>

class SendPacket : public ControlPacket
{
 protected:
   struct PacketSend {
      PacketType type;
      flow_t flowid;
      int shmkey;
      int datasize;
   };
   PacketSend *packet;
 public:
   SendPacket (unsigned short extrasize=0, char *buf=NULL);
   flow_t flowid ();
   void flowid (flow_t flowid);
   int shmkey ();
   void shmkey (int shmkey);
   int datasize ();
   void datasize (int datasize);
};

#endif
```

SharedMemBuffer Class

```
#ifndef SHMBUFFER_H
#define SHMBUFFER_H

#include "buffer.h"
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHMBUFFER_SHMSIZE (8196)

class SharedMemBuffer : public Buffer
{
 protected:
   int creator;
   int shmid;
   key_t key;
 public:
   SharedMemBuffer (key_t keytouse,
                int createnew=0, int
buffersize=SHMBUFFER_SHMSIZE);
   //SharedMemBuffer (SharedMemBuffer &rhs);
   ~SharedMemBuffer ();

   key_t Key ();

   //virtual SharedMemBuffer &operator =(SharedMemBuffer &rhs);
};

#endif
```

StartPingThroughputPacket Class

```
#ifndef STARTPINGTHROUGHPUTPACKET_H
#define STARTPINGTHROUGHPUTPACKET_H

#include "flow.h"
#include "controlpacket.h"

class StartPingThroughputPacket : public ControlPacket
{
 protected:
   struct PacketStartPingThroughput {
      PacketType type;
      flow_t flowid;
   };
   PacketStartPingThroughput *packet;
 public:
   StartPingThroughputPacket (unsigned short extrasize=0, char
*buf=NULL);
   flow_t flowid ();
   void flowid (flow_t flowid);
};

#endif
```

TerminatePacket Class

```
#ifndef TERMINATEPACKET_H
#define TERMINATEPACKET_H

#include "flow.h"
#include "controlpacket.h"
#include <unistd.h>

class TerminatePacket : public ControlPacket
{
 protected:
   struct PacketTerminate {
      PacketType type;
      flow_t flowid;
   };
   PacketTerminate *packet;
 public:
   TerminatePacket (unsigned short extrasize=0, char
*buf=NULL);
   flow_t flowid ();
   void flowid (flow_t flowid);
};

#endif
```

UDP Class

```
#ifndef UDP_H
#define UDP_H

#include "comm.h"
#include "buffer.h"
#include <netinet/in.h>

class UDP : public Comm
{
 protected:
   int sockfd;
   unsigned short port;
   struct sockaddr_in remoteaddr;
   struct sockaddr_in lastrecvaddr;    // last address from
which data was received
 public:
   struct Addr {
      char host[30];
      unsigned short port;
   };
 public:
   UDP (unsigned short localport);
   UDP (Addr &raddr, unsigned short localport=INADDR_ANY);
   UDP (char *host, unsigned short port, unsigned short
localport=INADDR_ANY);
   UDP (UDP &rhs);
   virtual ~UDP ();
   virtual int Send (Buffer &buffer);
   virtual int Send (Buffer &buffer, struct sockaddr_in *dest);
   virtual int Reply (Buffer &buffer);
   virtual int Receive (Buffer &buffer);
   virtual int NonBlockingReceive (Buffer &buffer);
};

#endif
```