ADAPTIVE FLOW MANAGEMENT OF MULTIMEDIA DATA WITH A VARIABLE QUALITY OF SERVICE

A thesis submitted in fulfilment of the requirements for the degree of

MASTER OF SCIENCE

of

RHODES UNIVERSITY

by

PAUL STEPHEN LITTLEJOHN

December 1998

Abstract

Much of the current research involving the delivery of multimedia data focuses on the need to maintain a constant Quality of Service (QoS) throughout the lifetime of the connection. Delivery of a constant QoS requires that a guaranteed bandwidth is available for the entire connection. Techniques, such as resource reservation, are able to provide for this. These approaches work well across networks that are fairly homogeneous, and which have sufficient resources to sustain the guarantees, but are not currently viable over either heterogeneous or unreliable networks.

To cater for the great number of networks (including the Internet) which do not conform to the ideal conditions required by constant Quality of Service mechanisms, this thesis proposes a different approach, that of dynamically adjusting the QoS in response to changing network conditions. Instead of optimizing the Quality of Service, the approach used in this thesis seeks to ensure the delivery of the information, at the best possible quality, as determined by the carrying ability of the poorest segment in the network link.

To illustrate and examine this model, a service-adaptive system is described, which allows for the streaming of multimedia audio data across a network using the Real-Time Transport Protocol. This application continually adjusts its service requests in response to the current network conditions. A client/server model is outlined whereby the server attempts to provide scalable media content, in this case audio data, to a client at the highest possible Quality of Service.

The thesis presents and evaluates a number of renegotiation methods for adjusting the Quality of Service between the client and server. An *Adjusted* QoS renegotiation method algorithm is suggested, which delivers the best possible quality, within an acceptable loss boundary.

Table of Contents

Table of Figures

Chapter 2 — Related Research

Chapter 3 - System Design

Chapter 4 - Implementation Issues

Chapter 5 - QoS Renegotiation Methods

Chapter 6 - Experiments and Results

Preface

I would just like to thank a few people who have contributed towards this thesis.

Professor Peter Clayton, my supervisor, for the constant encouragement he provided throughout the two years that I was privileged enough to work with him. It is always too easy to become obsessed with the small details when working on a. project such as this and Professor Clayton always managed to remind me to keep the "big picture" in view.

Professor Shaun Bangay, a living reference for any of my C++ related questions.

Jenny Hallowes for proof-reading this thesis. After doing the same job for my Honours project I am amazed that she agreed to do it again 1

Chapter 1

Introduction

The delivery of multimedia data across heterogeneous IP networks is a wide and varied field of research, presenting many challenges and posing numerous problems. Current IP networks, such as the Internet, do not possess sufficient available bandwidth to continuously deliver high-quality multimedia data streams.

To address the bandwidth scarcity, IP service providers and telecommunications operators are continually increasing the total bandwidth available to the broad range of IP applications. To improve the protocol support to multimedia traffic, several extensions to the IP[1] protocol (for example RSVP[2] Nankin *et. al.* 1997]) have been attempted, with the goal of providing network conventions that are more readily suited to delivering this type of data.

New classes of networking protocols, such as ATM[3] [ATM 1994], were designed with the specific goal of the guaranteed delivery of multimedia data. However, IP version 4[4] networks [ISI 1981] form the core part of the Internet, and widespread implementations do not provide for the provision of the delivery of a high QoS[5] of multimedia data across these networks.

Delivery of a guaranteed continuous QoS requires some form of network resource reservation. The packet-switched nature of IPv4 networks prevents predictions regarding the path that data packets will travel. A continuous stream of data packets may travel different paths from source to destination, and may consequently be received in a different order to which they were sent. Due to these characteristics, it is impossible to predict the load on an IP network segment at a specific point in time. Resource reservation, without the lower-level support of routers and switches, over the standard IP networks (version 4) is therefore not possible.

ATM and RSVP are two examples of network protocols that provide for resource reservation from server to client. Delivery of multimedia data streams, such as video, requires a large reservation of the

---

[1] IP — Internet Protocol

[2] RSVP - Resource Reservation Protocol

[3] ATM - Asynchronous Transfer Mode

[4] IPv4 — IP version 4

[5] QoS — Quality of Service

available bandwidth. The maximum guaranteed QoS is only as high as can be supported by the minimum bandwidth segment between the server and client. If delivery of a higher QoS is attempted, loss of information may occur across the minimum bandwidth segment.

Although resource reservation mechanisms guarantee the specified bandwidth for the server/client communication, a large proportion of this bandwidth may be wasted during periods of inactivity in communication between the proprietor server and client, and other network users are prevented from accessing this unused bandwidth.

Without any form of resource reservation, all users are competing for the same available network facilities. During periods of high network activity, the attempted delivery of a constant QoS results in high packet loss, and a corresponding lower delivered throughput. During multimedia sessions, applications often suffer quality degradation caused by network saturation or host congestion. In particular, network saturation may lead to a rapid decrease in QoS. When no resource reservation mechanisms are present, the delivery of data across heterogeneous networks requires the desired QoS levels to be tempered by the available bandwidth.

Schulzrinne identifies the delivery of bandwidth-intensive data as a potential problem area: "However, the current Internet cannot yet support the full potential demand for real-time services. High bandwidth services using RTPI, such as video, can potentially seriously degrade the QoS of other network services. Thus, implementers should take appropriate precautions to limit accidental bandwidth usage." [Schulzrinne *et. al.* 1996; 4].

In situations where host and network resources are scarce, or do not provide QoS guarantees, it is important to make efficient use of existing resources in order to accommodate end-user requirements. By adapting to fluctuations in available network resources, it is possible for distributed applications to deliver a meaningful information-content at a variable QoS level. A video application would therefore reduce the number of colours or the frame-size or frame-rate during times of high network activity, but the core content would still be transmitted. Instead of simply losing data packets, the QoS is reduced to prevent this from occurring. The delivery of the information-content is therefore guaranteed, whereas the QoS of the delivered information is not guaranteed.

This thesis investigates a client/server model using RTP and RTCP[2]. An adaptive flow system is proposed, which uses explicit feedback from the receiver to dynamically adjust the data flow based on available network resources. Adaptive systems require constant information regarding the current

[1] Real-Time Transport Protocol

[2] RTCP - Real-Time Transport Control Protocol

network resource-state in order to perform accurate flow modelling. The delivered QoS is adjusted dynamically throughout the lifetime of the connection.

To test this concept, the thesis describes the design and implementation of a point-to-point audio-streamer, with an approach to flow management which attempts to maximize the use of available resources by continually delivering the highest possible QoS. No resources are explicitly reserved, and a best-effort delivery of the negotiated QoS is attempted.

Apart from the obvious bandwidth constraints on QoS, the level of delivered QoS is determined by the nature of the renegotiation method used. A renegotiation method is an algorithm that measures the network throughput and packet loss, and then increases, decreases or maintains the QoS of the delivered data stream. A number of renegotiation methods are proposed and tested using the audio-streaming application, with the objective of recommending an efficient renegotiation method for making the adaptive flow system viable within the test environment.

The thesis is structured as follows:

- Chapter 2 surveys the primary research in the published literature that is related to the networking, Quality of Service, and audio streaming issues of the RTP Audio Application described in this thesis.
- Chapter 3 describes the design of the entire system and introduces relevant QoS issues. A detailed discussion of the layered approach to solving the problem, including integration and interaction between different layers, completes the chapter.
- Issues relating to the implementation of the system are discussed in Chapter 4, which incorporates discussions of the communication and synchronisation aspects of the network and application levels.
- The renegotiation methods proposed and implemented to support the adaptive flow model are investigated in Chapter 5. This chapter includes a detailed description of each algorithm used.
- A series of experiments and the analysis of their results are presented in Chapter 6.
- Chapter 7 outlines possible future directions and extensions to the system, as exposed by this research.
- Chapter 8 concludes the report by summarising its contributions, and by assessing the overall usefulness of the approach.

Chapter 2

Related Research

2.1. Introduction

Much current research involves attempting to deliver a constant, guaranteed QoS, both in the end-systems and across the network resource. The main disadvantage of this approach is that the QoS must be guaranteed at every point along the network, from the server to the client. Across an heterogeneous IP network such as the Internet, this is not possible. Quality of Service issues relating to the delivery of a non-guaranteed QoS, as well as applications that have implemented an adaptive-flow technique for the delivery of multimedia data, are discussed in this chapter.

The approach proposed in the design section (Chapter 3) describes an adaptive-flow mechanism allowing the delivered QoS to adapt to the available network resource. Networking issues are related to the delivery of data- and control-packets from the server to the client. Currently, RIP and RTCP are the primary network protocols utilised for this purpose. The advantages and disadvantages of the protocol in relation to the system discussed in the following chapters is the presented in section 2.2.

The final component of the system, audio streaming, is implemented using the OSS[1] model. A discussion of the OSS services, including a description of the different audio sampling parameters that are required for the RT? Audio Application discussed in chapter 3, completes the Related Research section.

2.1.1. Real-Time Transport Protocol

2.1.1.1. Introduction

The Real-Time Transport Protocol is designed for use with multicasting multimedia applications [Schulzrinne *et. al.* 1996]. Although the primary objective of this thesis is not to develop a multicasting application, there are many features of RIP that make it very attractive as a point-to-point application development protocol. RTP is a suitable protocol for applications transmitting real-time data, such as

[1] OSS — Open Sound System

audio or video. The other suitable application categories include storage of continuous data, interactive distributed simulation, and control and management applications [Schulzrinne *et. al.* 1996]. The RTP Audio Application is an example of the control and management class of application.

2.1.1.2. Protocol Specifics

A full discussion of the RTP and RTCP protocols is beyond the scope of this document and the discussion is limited to portions of the protocol that are relevant to the RTP Audio Application to be discussed in the following chapters.

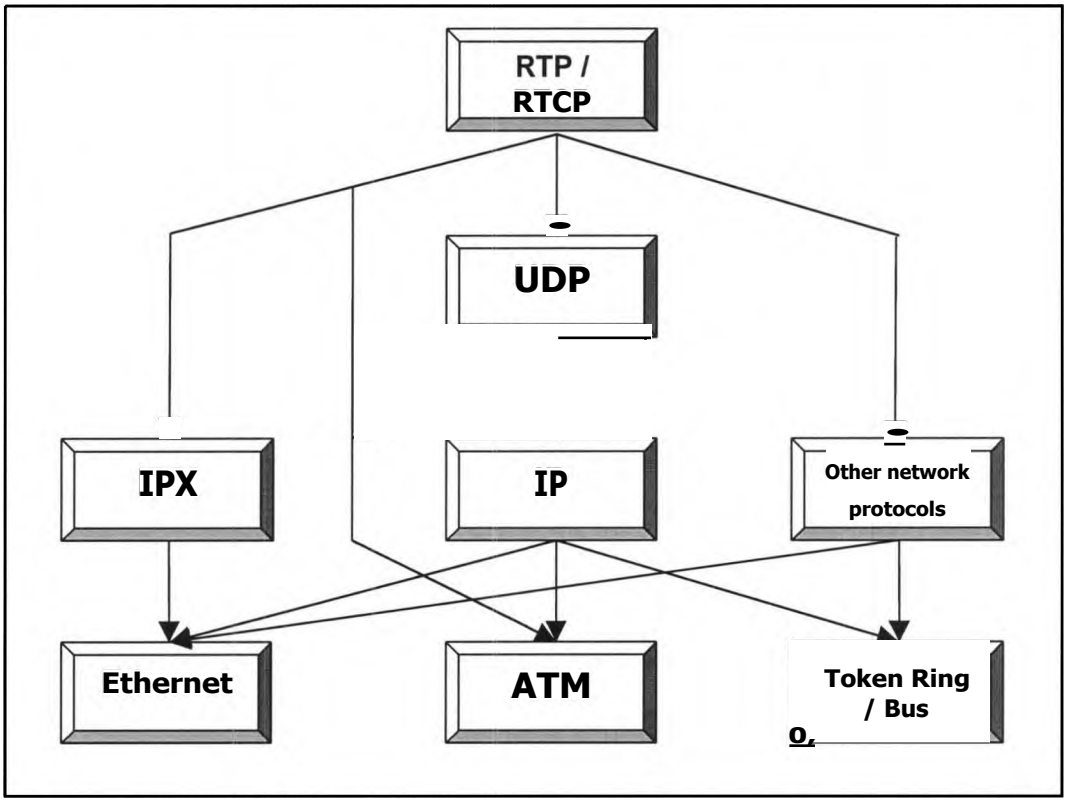

Figure 2.1. - RTP in relation to other networking protocols [Schulzrinne 1994]

Figure 2.1 clearly illustrates the position of RTP as a network transport protocol [Schulzrinne 1994]. RTP can be implemented over a number of different transport protocols (IPX, UDP, ATM etc.) which then provide the necessary transport mechanisms for the delivery of the data.

RTP packets are created without regard to the underlying transport layers or protocols. Although RTP provides framing mechanisms for encapsulation of the multimedia data into RTP packets, it is not responsible for the implementation of transport layer mechanisms such as routing. RTP applications

generally use UDP1 as the transport protocol. UDP provides checksum and multiplexing capabilities, as well as using the well-known Internet Protocol[2] for the packet delivery mechanisms.

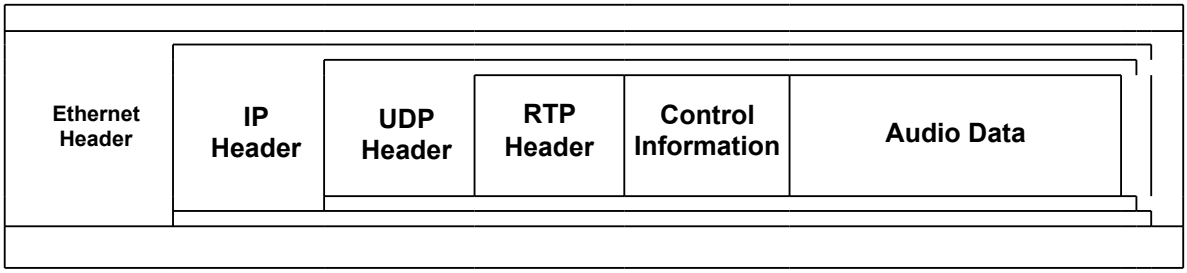

Figure 2.2. - Encapsulation of the RTP packet including the IP and UDP headers

There is no defined maximum length for a Real-Time Transport Protocol packet. The maximum packet length of the underlying protocol determines the amount of data per RTP packet. An RTP packet has no length field and cannot be spread over numerous lower-level transport packets. A requirement of RTP is that the data and headers be transported within a single data packet, as reconstruction of separate RTP packets is not possible [Schulzrinne *et. al.* 1996].

2.1.1.3. RTP Fixed Header Fields

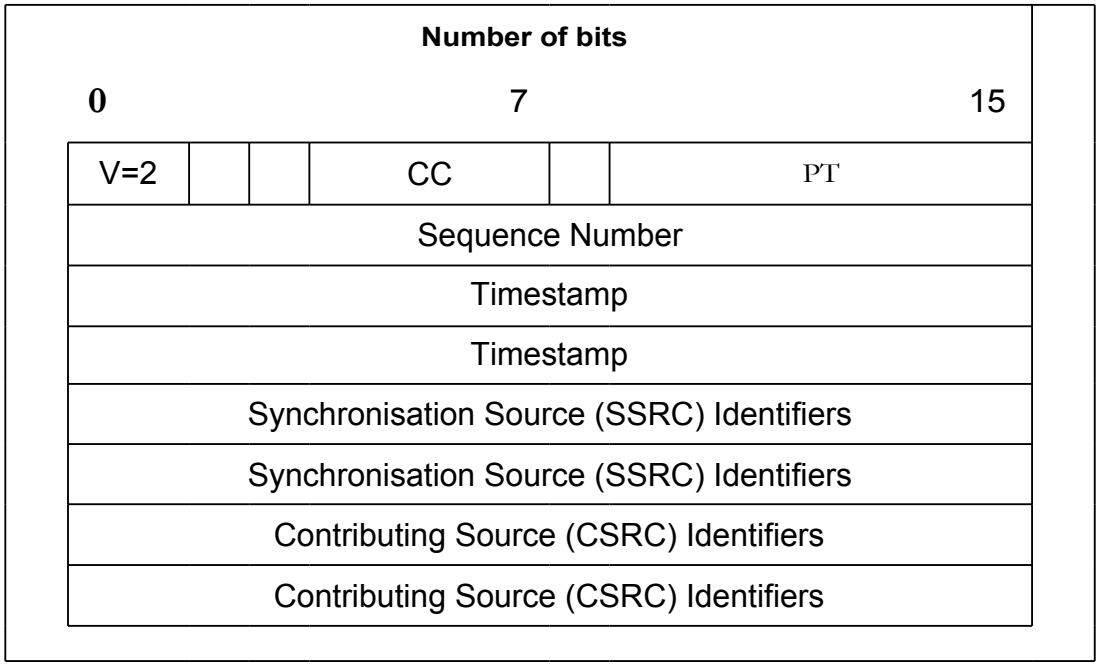

Figure 2.3. - RTP Header fields [Schulzrinne *et. al.* 1996]

I UDP — User Datagram Protocol

[2] IP — Internet Protocol

Figure 2.3 shows all of the RTP header fields. Because the RTP Audio Application does not utilise all available RTP header fields, this discussion is limited to only the required header fields. It is necessary to note, however, that the RTP implementation includes all non-multicasting fields. The CSRC1 identifier is limited to a single source for unicasting applications. The RTP header fields discussed below are defined by Schulzrinne *et. al.* [1996].

Payload Type (PT) [7 bits]

This field identifies the format of the RTP payload and determines the media-encoding to be used by the application. A profile specifies a default static mapping of payload type codes to payload formats.

Sequence Number [16 bits]

Increments by one for each RTP data packet sent, and may be used by the receiver to detect packet loss and restore the packet sequence. The initial value of the sequence is random.

Timestamp [32 bits]

The timestamp is the sampling instant of the first octet in the RTP data packet. The sampling instance must be derived from a clock that increases linearly in time in order to allow synchronisation and jitter calculations. The clock resolution must be sufficient for the desired synchronisation accuracy and for measuring packet arrival jitter. The initial value of the timestamp is random.

SSRC[2] [32 bits]

The SSRC field identifies the synchronisation source. This identifier is chosen randomly with the intention that no two synchronisation sources within the same RTP session will have the same SSRC identifier.

CSRC list [0 to 15 items, 32 bits each]

The CSRC list identifies the contributing sources for the payload contained in the packet. The number of identifiers is given by the CC[3] field. If there are more than 15 contributing sources, only 15 are identified.

---

[1] CSRC — Contributing Source

[2] SSRC — Synchronising Source

[3] CC - Contributing Count

2.1.1.4. Protocol Advantages

As described in section 2.2.1.3, RTP packets contain a sequence number and a timestamp. Real-time applications require a correct ordering of data packets. However, packet-switched networks do not guarantee delivery in a sequential order. RTP has therefore provided a sequence number field that allows the receiver to reconstruct a stream of data in the correct order. If an RTP packet arrives out of order, the RTP sequence number for the packet will be lower then the preceding packet and will therefore not be played. Consequently, sequence numbers can be used to detect losses in a data stream and will also ensure that packets are delivered in the correct order.

The RTP timestamp is used to place the incoming audio and video packets in the correct timing order. Synchronisation of different media streams is also achieved via the timestamp. For example, a video conferencing application requires an audio stream and a separate video stream. Lip-synching and other synchronisation issues are achieved by comparing timestamps on different RTP packets and determining the exact playout time of each data segment. Timestamps are used in the adaptive flow system discussed in the following chapters to calculate the throughput of the data stream at the receiver.

RTP packets include a payload type field, which is used to map particular payload types to specific media-encodings. The payload types identify the type of data that is being delivered within the RTP packet. A complete RTP implementation requires a profile specification document defining the payload type codes and also a payload format specification document defining how each payload type is to be transported by RTP [Schulzrinne *et. al.* 1996].

2.1.1.5. Protocol Disadvantages

RTP was chosen as the transport protocol for the RTP Audio Application due the flexibility of the protocol as well as the timestamping, sequence numbering and payload type identification features. There are, however, various disadvantages of the protocol that have influenced the design and implementation of the RTP Audio Application prototype.

RTP does not provide for any form of reservation of the available bandwidth. As previously mentioned, resource reservation enables an application to guarantee a specific amount of bandwidth between the server and client. Across a heterogeneous IP network, such as the Internet, resource reservation is not possible unless a protocol such as ATM or RSVP is implemented. The lack of resource reservation is, therefore, a disadvantage for applications requiring a guaranteed QoS. An

adaptive flow system of data delivery, such as that implemented in the RTP Audio Application, does not require a guaranteed QoS and the lack of resource reservation is therefore not a protocol disadvantage in this situation.

There is, therefore, no guarantee of a specific QoS for real-time services. Applications attempting to deliver a continuous QoS will experience packet loss during times of high bandwidth usage. Similarly, a large available bandwidth is underutilised, as the application is unable to increase the delivered QoS.

Delivery of data packets is dependent on the underlying transport protocol. In an IP packet-switched network environment, data packets may take a number of different routes from the server to the client. Unless the underlying transport protocol guarantees sequential packet delivery, RTP does not guarantee in-order packet delivery. The sequence number field in the RTP packet is used to reconstruct data streams and RTP does not need to assume a sequential packet-delivery.

2.1.2. Real-Time Transport Control Protocol

The Real-Time Transport Control Protocol is used not only for the monitoring and control functions that are normally associated with the protocol, but also for the dynamic negotiation of the QoS between the client and the server. After monitoring the delivered packet rate, the client periodically informs the server (by means of an RTCP packet) of the new QoS value to be delivered.

According to Schulzrinne *et. al.* [1996], RTCP performs four functions:

1. To provide feedback on the quality of the data distribution. This relates to the flow- and congestion-control functions of other transport protocols. Control packets in multicasting applications perform the additional task of diagnosing distribution faults, such as determining whether errors are global or restricted to a single receiver.
2. RTCP carries a canonical name, or CNAME, for an RTP source. The CNAME is required to keep track of each participant. It is also used to associate multiple data streams from a single participant in a set of related RTP sessions.
3. To facilitate scaling in a multicasting environment, the delivery rate of the control packets needs to be regulated.
4. The final function allows session control information to be conveyed to all participants. The user interface then displays this information.

The RTCP implementation for the RTP Audio Application provides only a limited set of features. No multicasting functions are required and RTCP is therefore limited to providing feedback from the data receiver to the server. The feedback received is used to perform the dynamic QoS adjustments.

2.1.3. RTP and RTCP Multicasting

According to the Mbone Information web [Icast], "IP Multicast facilitates distributed applications to achieve time-critical "real-time" communications over wide area IP networks through a lightweight, highly threaded model of communication. Data is distributed and replicated via a series of multicast routers to their destinations as opposed to individual hosts. IP-Multicast is the class-D addressing scheme in IP and has been allocated the IP address range from 224.0.0.0 to 239.255.255.255."

RTP was originally designed as a multicasting transmission protocol (as long as the underlying transport protocol supports multicasting). Although the RTP Audio Application discussed in this thesis does not support multicasting, it may be added at a future stage and is therefore worth discussing here.

According to Schulzrinne *et. al.* [1996], the delivery of RTCP packets across a multicast network is useful for three reasons:

1. Monitoring the data-receive rates of all participants. In a multicasting environment (especially the Internet) it is unlikely that all participants will be able to receive data at the same data rate. Monitoring the data-receive rate of all participants in order to determine whether it is possible to increase or decrease the transmission rate is therefore necessary.
2. Checking whether a network is able to receive transmissions. If a receiver that wishes to receive a multicast data transmission is able to deliver an RTCP data packet to a multicast address, then it follows that the receiver will also be able to receive the multicast data transmission.
3. If participants are not delivering RTCP reports, then it is possible to determine whether the error is local, regional or global.

Scalability of the control-packet delivery rate must be controlled in a multicasting environment. If control packets in a multicasting environment are delivered at a constant rate by all session members, then the amount of bandwidth required will increase linearly with the number of participants. As the number of participants increases, the transmission rate of the control packets must decrease correspondingly. However, in an unicasting environment the control packets are delivered at a constant rate, and this problem does not arise.

2.1.4. RTP and RTCP Security

Until lower-layer protocols are able to provide for all the security needs of an RTP application, the confidentiality services need to be implemented in the RTP application itself. Encryption of packets ensures a higher confidentiality of the transmitted information. A detailed description of encryption algorithm is not discussed, as encryption has not been implemented in the test system developed for this thesis.

2.2. Quality of Service

A vital component of adaptive-flow delivery systems is whether an acceptable QoS can be maintained. QoS issues relate not only to application-specific media parameters, such as frame-rate and frame-size for a video application, but also to timing constraints and resources required to provide the delivered QoS.

An investigation of general QoS issues and their relevance to the system proposed in this thesis is presented, and a number of applications that attempt to provide an adaptive-flow data delivery mechanism are also discussed.

2.2.1. Quality of Service Issues

Although the RTP Audio Application investigated throughout this thesis does not provide a guaranteed QoS; various QoS issues in both the end-system and network resource need to be explored. According to Aurrecoechea: "For applications relying on the transfer of multimedia, and especially continuous media flows, it is essential that the QoS is configurable, predictable and maintainable system-wide, including end-system devices, communications subsystems and networks". These issues are relevant to all applications delivering multimedia data. As has already been elaborated upon, an adaptive flow system with adjustable QoS does not attempt delivery of a constant QoS for the entire lifetime of the connection.

The delivery of a guaranteed QoS is an end-to-end application issue. QoS assurances should apply to the complete flow of media from the remote server across the network and finally to the point of delivery. Not only must sufficient network bandwidth be reserved for the duration of the connection, but resources in the end systems must also guarantee timely delivery [Campbell *et. al.* (1)]. Although necessary end-system resources are negligible in an application such as the RTP Audio Application, for

a large-scale video scheduling system these demands can become quite extensive. Media devices, the operating system and thread scheduling are only some of the aspects of the system that need to be investigated to guarantee a specified QoS.

The delivery of multimedia data, as opposed to traditional data such as executable files, presents numerous unique problems. Multimedia is characterised by continuous media, such as audio, video and graphical animations. Not only is a greater strain placed on communications media to deliver this data, but also different media require different levels of jitter, error control and packet loss prevention in order to maintain the required levels of service [Busse *et. al.].* For example, highly compressed audio does not withstand fluctuations in quality as well as an uncompressed video stream can. Occasional dropped frames in a video sequence are noticed less than lost portions of an audio stream. Although not directly related to the specific QoS, delay jitter must be kept at the lowest level possible. High jitter results in many late packets and gaps occurring during the playout of a data stream.

The quality of delivered multimedia data is directly proportional to the bandwidth and the encryption method used. Assuming that the encryption method is constant throughout encodings, the higher the QoS required, the proportionally higher the required bandwidth will be. There is, therefore, a tradeoff between the required QoS and the available bandwidth.

2.2.2. QoS-Architectures

As already discussed, the delivery of a guaranteed QoS is only possible if the guarantee is applicable across all end-system layers as well as the connected network resource. Much research is currently focused on the provision of a QoS-Architecture [Campbell *et. al. (1)].*

The QoS-Architecture requires the delivery of the QoS to be incorporated across all layers of the system, and not added to existing systems on a piecemeal basis. The mapping of QoS details between layers is simply to protect users from the underlying communications details. Management functions and QoS support mechanisms are included as core parts of the system. These functions control end-to-end QoS negotiation and admission control, policing of the negotiated QoS to ensure users do not violate negotiated QoS parameters, and also monitoring to ensure the ISP[1] is maintaining the negotiated QoS levels.

ISP — Internet Service Provider

2.2.3. Quality of Service Applications

A number of separate QoS applications are investigated. All of the applications do nor attempt to deliver a guaranteed QoS. All applications that are discussed in this section are related to the aspects of the RTP Audio Application that is discussed in the following sections.

2.2.3.1. QoS Algorithms

An algorithm for the renegotiation of the QoS is proposed by Busse *et. al.* The current congestion of a network is calculated according to the smoothed value of the packet loss rates. Based on this metric, the congestion-state (as seen by the receivers) is determined and the system is considered to be in an unloaded, loaded, or congested state. A linear regulator with dead-zone then adjusts the bandwidth. For a point-to-point network connection, i.e. an unicast environment, the congestion-state is directly mapped to decrease, hold, or increase the delivered bandwidth. The congestion-state for a multicast environment is more difficult to determine, however, and is not related to this work. The unicast algorithm is implemented in the RTP Audio Application and is discussed in detail in section 5.4.4.

The QoS adaptation algorithm proposed by Campbell *et. al.* attempts to guarantee the delivery of a base layer of an MPEG-2 video stream and provide an adaptive flow system to enhancement layers [Campbell *et. al.* (2)]. The base layer undergoes a full end-to-end admission control test, but enhancement layers are admitted without any such tests. Enhancement layers are rate-controlled, based on explicit feedback about the current state of the on-going flow and the availability of residual bandwidth. This algorithm requires the guaranteed delivery of the base-layer and this prevents the implementation of this algorithm in the RTP Audio Application.

2.2.3.2. QoS Control Mechanisms

A series of QoS control mechanisms is proposed for the delivery of real-time traffic control flows. These control mechanisms are implemented in conjunction with the QoS-Architecture [Aurrecoechea *et. al].*

The following control mechanisms are proposed:

1. Flow shaping - regulates the flows based on user-supplied flow performance specifics.
2. Flow scheduling - manages the forwarding of flows in end-systems and networks in an integrated manner.

3. Flow policing - observes whether the QoS contracted by a user is being adhered to.
4. Flow control - differentiates between open- and closed-loop flow control. Applications using closed loop based protocols must be able to adapt to fluctuations in the available resources.
5. Flow synchronisation - required to control the event ordering and precise timing of multimedia interactions.

The requirement of a Q0S-Architecture prevents these control mechanisms from being implemented within the RTP Audio Application.

These control mechanisms have, however, been implemented in other systems:
Within the Q0S-Architecture, the Transport System comprises a number of QoS control mechanisms. The flow scheduler provides appropriate rate-control to ensure per flow bandwidth guarantees. Open loop flow control, based on a token bucket scheme, is provided by the flow shaper [Campbell *et. al.* (2)].

A set of QoS control mechanisms is proposed by Alfano and Radouniklis, which enables the dynamic adaptation of application parameters depending on the user requirements and resource status [1996]. These control mechanisms are implemented within the RTP Audio Application and are discussed in detail in section 5.4.6.

2.2.3.3. Dynamic Feedback

The QoS adaptive algorithm proposed by Busse *et. al.* requires dynamic network feedback based on the bandwidth requirements of multimedia applications. RTCP receiver reports are used to continuously compute the packet loss. Across a multicast network, a record for each receiver is maintained which contains the most recent receiver reports, including information related to session descriptor packets, the loss rate and the packet-delay jitter. The feedback reports generated are implemented as part of the RTCP protocol, and a similar feedback mechanism is implemented within the RTP Audio Application. A detailed description of the RTCP feedback mechanism is presented in section 3.3.2.2.

2.2.3.4. Service Commitment

The Integrated QoS for Multimedia Communications attempts to partition the level of service commitment into a small number of fixed levels, rather than as a continuous scale [Campbell *et. al.* 1993].

The following partitions are proposed:

1. Deterministic - The highest priority service possible. The QoS is guaranteed for hard real-time performance applications.
2. Probabilistic - QoS degradation occurs from time to time because of the statistical nature of network service. This service is suitable for applications delivering media types with a built in redundancy.
3. Best-effort — The lowest priority service. No network resources are allocated or monitored and only the available network resources are utilised.

The deterministic and probabilistic service commitments require QoS guarantees to be successful and are therefore not implemented within the experimental system described in the following chapters. The best-effort level of service commitment is implemented with a continuous QoS scale describing the various QoS levels.

2.2.3.5. QoS Protocols

A QoS Negotiation and Resource Reservation Protocol is proposed for negotiating the QoS and reserving resources for distributed application components and communications links. The protocol requires strict balancing between the QoS specified by the client, the resource capabilities of distributed system, and the functional capabilities of the distributed application. Although the protocol provides the QoS negotiation functions required for the RTP Audio Application, the resource reservation functions require end-to-end implementation at all network components. This prevents implementation across heterogeneous IP networks and as a result, the protocol has not been used [Dermler *et. al.* 1995].

2.2.3.6. Experiments and Results

Busse *et. al.* implemented the QoS adaptive algorithm and conducted a series of experiments using the vi c video conferencing application. The experiments were conducted using both an IP network (Internet) and an ATM network. The ATM network exhibits different loss characteristics to the IP network, and different controller parameters were used.

The results of the IP network experiments show that the controller parameters require accurate setting in order for the experiments to be successful. A comparison between packet loss with bandwidth control, and that without, illustrates the effectiveness of the proposed algorithm. Without bandwidth control, heavy losses ranging from 20%-50% were measured, but with bandwidth control, this value was maintained below 10%.

2.2.3.7. Conclusion

A discussion of research related to the RTP system is presented in this chapter. Although the research is not related directly to the delivery of a dynamically adjustable QoS, many aspects of the system are relevant in the design of the RTP Audio Application and will be discussed in more detail in the system design chapter (chapter 3).

2.3. Media Services

In the Quality of Service Issues section (2.3.1.), it was highlighted that different multimedia data types exhibit different QoS characteristics. Video streams exhibit different QoS characteristics to audio streams, as well as having different loss and throughput constraints. Digital audio is the only data type that has been implemented in this test system and all references to media devices relate to this media service. Video data is therefore no longer investigated.

The background information of the digital audio data and the detailed description of the OSS service, including all corresponding sampling and control parameters, are part of the OSS documentation from 4 Front Technologies [4Front].

2.3.1. Digital Audio Data

Digital audio data is a sequence of samples taken from the input sound stream at constant time intervals. The volume of the input stream at each sampling interval is represented in the sample. Digital audio is a combination of the sampling rate (measure of the highest frequency that the sample records), the number of channels (mono or stereo) and the quality of the sample (either 8-bits or 16-bits). The sampling encoding determines the dynamic range of the recorded signal. The maximum dynamic range of the recorded signal is equal to the number of bits * 6 dB. An 8-bit sampling resolution therefore enables a dynamic range of 48 dB and 16-bit resolution provides for 96 dB [4Front].

The disk space that is required to store the audio data is dependent on the sampling rate, sampling encoding and the number of channels. The number of bytes required to store one second of audio data is calculated as (sampling rate * sampling encoding * number of channels). Stereo audio data therefore requires twice as much storage space as the same data that is recorded in mono. Correspondingly, delivery of digital audio data across a network places similar strains on network resources. The higher the quality of data that is required, the larger the amount of bandwidth that is needed to deliver it. A

tradeoff therefore occurs between the QoS of the data required, and the bandwidth that is available to deliver it.

2.3.2. Open Sound System

Open Sound System (OSS) is a device driver for accessing sound cards and other sound devices using various UNIX operating systems [4Front]. There are various sound devices that are available with OSS, including the digitised voice device, mixer device, synthesiser device and a MIDI interface. The device that has been extensively utilised in this research is the digitised voice device. The other devices have not been used and will not be discussed further.

One of the main goals of the OSS API is full portability of applications. Portability not only refers to the program's ability to work on different machines operating with different operating systems, but it also refers to the application operating on different sound hardware. The OSS API attempts to hide access to the specific underlying hardware layers.

The audio types supported by OSS include: digital audio sampling and playback, MIDI, electronic music, streaming audio, speech recognition/generation, computer telephony and synchronised audio capabilities.

In the normal UNIX manner, each sound device has a file device associated with it. For example, the mixer device is associated with /dev/mixer . Similarly, the digitised voice device is associated with /dev/dsp. Implementation of the audio services has been completed with /dev/dsp and any mention of a sound device will refer to this device by default.

The sound device is a normal UNIX file device, and I/O control calls can be used to access the device. Reading to and writing from the sound card is achieved via the normal read ( ) and write ( ) device calls. Similarly, the device can be opened and closed for access by calling open ( ) and close ( ) . After a device is opened, default settings (of a very low quality) are set. Very simple access to the sound card can, therefore, be achieved by use of these calls. In order for more advanced functions to be performed, it is necessary to make use of ioct I ( ) function calls. These calls will be discussed in more detail in section 4.3.3.

2.3.2.1. Sound Control Parameters

The order in which the parameters for the sound device are set is important. Firstly, the sampling encoding (number of bits) must be set, followed by the number of channels (mono/stereo) and finally, the sampling rate (speed). The OSS documentation [4Front] explains that the prescribed order is due to limitations in certain hardware, which may only be able to produce sound at 44.1 kHz mono and at 22.05 kHz stereo. If the device tries to set the sampling rate at 44.1 kHz then once the number of channels has been set to stereo, the sampling rate is changed to 22.05 kHz. However, the user will not be aware that the sampling rate has been altered.

It is necessary to note that the sampling parameters may only be set between the time that the device is opened and the first read (), write () or other ioct 1 () calls.

a. Sample Format

The sample format determines the quality of the audio. The implementation of OSS in the RTP Audio Application requires the use of AFMT_U8 and AFMT_S16 LE for 8 bit and 16 bit sampling sizes respectively. Other formats could be implemented for a higher degree of portability amongst different sound cards and operating systems [4Front].

b. Number of Channels

The number of channels can be set by initialising the SNDCTL DSP STEREO parameter to a value of 1 (stereo) or 0 (mono). Alternatively, SNDCTL_DSP_CHANNELS can be used to set the number of channels to 1 or 2. If the device does not support stereo mode, then it will automatically set the number of channels to 1 [4Front].

c. Sampling Rate

OSS permits a range of sampling rates from 1 Hz to a maximum of 2 GHz. Individual sound cards do, however, set upper and lower limits on the permissible range of sampling rates. As will be described in the System Design chapter, three different sampling rates have been implemented in the RTP Audio Application, namely: 11025 kHz, 22050 kHz and 44100 kHz. The maximum rate of 44100 kHz was decided due to the fact that audio from a CD player was transmitted, and this is the maximum sampling rate of a CD player [4Front].

d. Other Control Parameters

As has already been mentioned, sampling parameters can only be altered between opening the device, and the first read ( ) or write ( ) or other ioct1 ( ) call. The following i oct1 ( ) calls are used to reset the sound device and accept new sampling parameters:

1. SNDCTL_DSP_SYNC is used when the application must wait until the last byte is written to the audio device and then control is returned to the calling program. This call can result in a delay of several seconds, depending on the amount of data in the buffer that must be output.
2. SNDCTL_DSP_RESET stops the device immediately and control is returned to the calling program. This call is also used when the last recording call to the device has been made and the device is not going to be closed immediately.
3. SNDCTL DSP POST is similar to SNDCTL_DSP_SYNC, except that the device is informed that there is likely to be a pause in the output of the data. This enables the pause to be managed more intelligently. This call is generally used when there will be a relatively long break before the next sample is output.

If the sampling parameters are to be changed, then either SNDCTL_DSP_SYNC or SNDCTL_DSP_RESET should be used. If the device is to be switched from recording mode to playback mode, then one of these two calls must be used [4Front].

2.4. Conclusion

Use of a QoS-Architecture for the RTP Audio Application (discussed in chapter 3) is not possible due to the primary requirements of the system being able to operate across a heterogeneous IP network. In order to minimise end-system requirements, it is also decided not to utilise end-system resources guaranteeing a specified QoS. This design decision is based on the fact that if it is not possible to guarantee the delivered QoS across the network component then it is pointless attempting to guarantee the QoS in the end-system.

The related research presented in this chapter presents background information required to understand the system design concepts presented in Chapter 3. Networking issues, including the RTP and RTCP protocols, are implemented within the Resource Layer of the RTP Audio Application. The QoS issues discussed are not implemented as part of a specific system layer but relate to overall design issues. The media service is implemented as part of the Application Layer. The dynamic adjustment of the media sampling parameters results in an audio stream with varying QoS.

Chapter 3

System **Design**

3.1. Introduction

In the preceding chapters, the significance of the variable QoS, point-to-point solution pursued by this thesis was explored. Chapter 3 provides a synopsis of the overall design of a test system that was used as a prototype in this thesis. This chapter describes the layered approach taken in the designing of the audio streaming application, and justifies the choice of the RTP and RTCP protocols as the base infrastructure for supporting this kind of communication. An analysis is presented, outlining the way in which design decisions translate into QoS characteristics, and how the various layers implement QoS requirements.

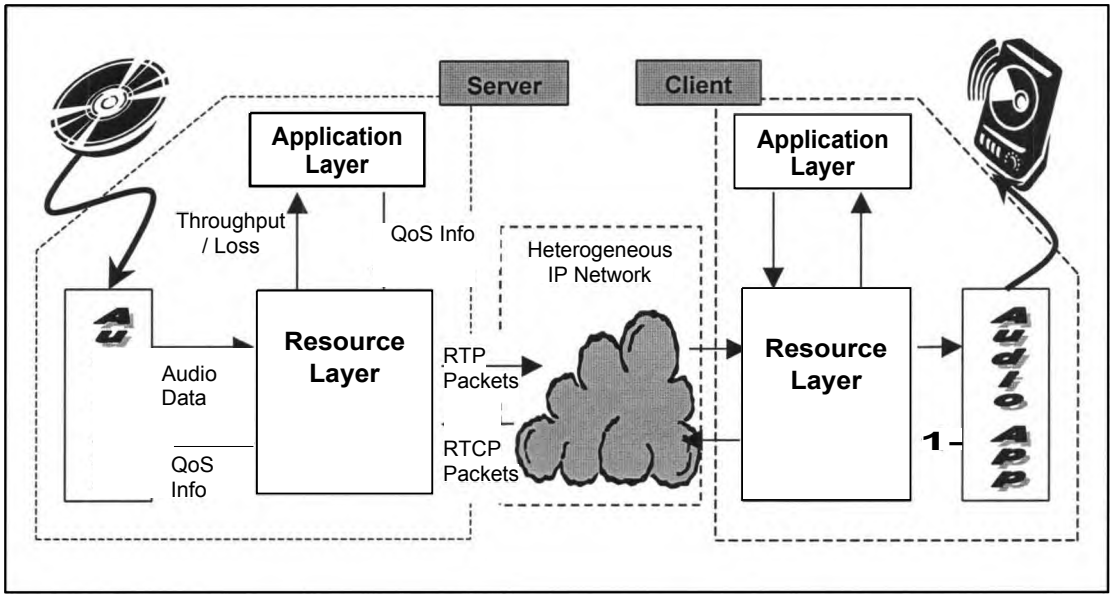

Figure 3.1. - RTP server/client audio application

Figure 3.1. illustrates the client/server nature of the RTP Audio Application. The renegotiation process required for the adaptive flow delivery of the audio data is the core component of the system. A detailed discussion of the renegotiation process and the interactions between system layers is presented in the following chapters.

The design of the RTP Audio Application has been divided into three separate layers based on the approach proposed by Alfano and Radouniklis [1996]. The Resource Layer is responsible for all

networking aspects of the system. The real-time nature of the renegotiation of the data flow, and the QoS issues arising from the renegotiation process, are handled by the Application Layer. Interactions between the user and application are the responsibility of the User Layer.

Although logically distinct and separate, a large amount of interaction between the layers must occur in order for the system to operate efficiently. This interaction occurs in the form of vertical and horizontal integration. Vertical integration refers to the correspondence between different layers of the system, and horizontal integration deals with the interdependence of sections within a particular layer.

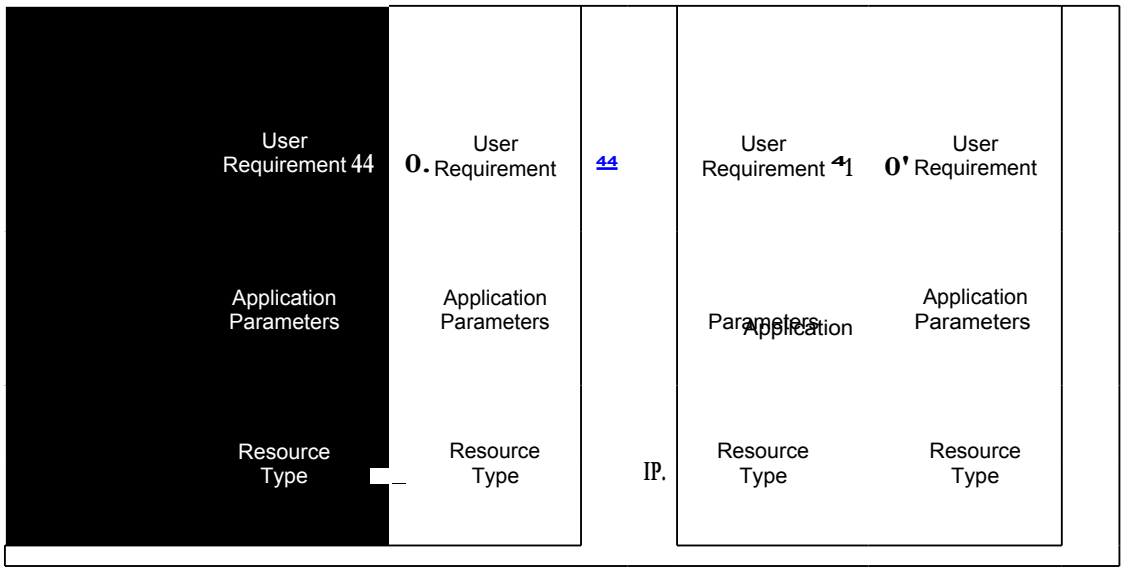

Figure 3.2. - Interaction between different layers and within a layer of the system[Alfano and Radouniklis 1996]

3.1.1. Vertical integration

Vertical integration enables interrelation between the different layers of the system, whilst allowing each layer to be a logically separate entity. Connection between the different layers occurs by introducing mapping mechanisms. Mappings allow information to be transferred and also transformed between layers of the system, while not losing any meaning to the new layer.

User QoS requirements from the User Layer are translated to media service parameters in the Application Layer. After altering the media device with the media parameters, the latter are then transferred from the Application Layer to the Resource Layer as resource QoS requirements. The Resource Layer is then able to deliver the resource requirements across a network to form a network integration layer with the client. A mapping table containing relevant QoS attributes is provided for

each media service. The mapping table is used as a look-up table if any transformation of a QoS parameter is required (figure 3.4).

3.1.2. Horizontal integration

Horizontal integration occurs exclusively within a specific layer of the system. The User Layer enables the user to specify quality requirements and other necessary parameters required by the system.

Integration of the media services into the Application Layer provides the mechanism for handling these services as part of the system as a whole, rather than as a separate entity. Knowledge of the state of the media services allows the control of these services to be handled more efficiently and effectively.

Horizontal integration within the Resource Layer provides mechanisms for the transparent delivery of data, regardless of the underlying networking protocols. For the RTP Audio Application, RTP and UDP are interdependent and the horizontal integration in the Resource Layer is concerned with the integration of these transport protocols.

3.2. Resource Layer

The design of the RTP Audio Application system requires a transfer of data from the server to the client, as well as delivery of control packets from the client back to the server. The Resource Layer is concerned primarily with the function of transmitting and receiving these packets. The design of this layer requires the interaction of various networking protocols to provide for the real-time delivery of multimedia content.

The Resource Layer uses RTP for the delivery of the data packets. Coupled with the RTP is RTCP, which is primarily concerned with the monitoring and control of RTP packets that are passed between the server and the client. RTCP is related to the flow- and congestion-control mechanisms of other transport protocols [Schulzrinne *et. al.* 1996]. As has been discussed in Chapter 2, the higher-level network protocols (RTP and RTCP) require a lower-level transport protocol for the actual delivery of the data and control packets.

3.2.1. Real-Time Transport Protocol

RTP is intended to be flexible in order to transport specific information required by a particular application. For this reason, RTP is often integrated into the application processing rather than maintained as a separate transport layer. The design of the RTP Audio Application has, however, separated the data processing functions from the delivery and transport mechanisms. The RTP Audio Application uses the RTP packet information that is delivered by the Resource Layer and passes it to the Application Layer for processing.

This enables a complete separation of the Resource and Application layers to occur. RTP is therefore included as part of the Resource Layer, and the information that is transmitted within the RTP and RTCP packets is processed separately by the Application Layer. Although RTP can be implemented as part of the Application Layer to provide data directly to the application, the RTP Audio Application is designed with a tight coupling between RTP and the lower-level transport protocol at the Resource Layer. The existing RTP implementation, used in the RTP Audio Application, is based on a tight coupling at Resource Layer and it was therefore not possible to implement RTP within the Application Layer.

The Resource Layer functions described in this section are based on an existing RTP implementation, described in Chapter 4. System design decisions were therefore decided upon with this existing system in mind.

The primary function of the RTP Audio Application system, which is the adaptive flow management of the multimedia data stream, is implemented within the Application Layer of the system. The data required for this function is, however, provided via the Resource Layer functions. Every RTP packet is delivered with a timestamp, sequence number and payload-type. Not only is the timestamp required for the reconstruction of the original data stream, but also to calculate the throughput for the period. The sequence number function is also two-fold. It maintains the original packet-order of the data stream, as well as determining packet loss for the renegotiation period. Payload-types are not an integral element in the design of the system discussed here, but if more multimedia data types are added to the system in future, it would be possible to differentiate between data types based on this RTP header field. These RTP header fields, namely the timestamp, sequence number and payload-type, provide the necessary information required for the higher-level processing of the multimedia data.

3.2.2. Real-Time Transport Control Protocol

RTP and RTCP are both abstracted from the underlying transport protocol. Despite the suggestion that the same underlying protocol should be used for RTP and RTCP [Schulzrinne *et. al.* 1996], it was decided that the RTP Audio Application should use UDP as the transport protocol for delivery of RTP data packets and TCP/IP for delivering RTCP packets. This decision was made because TCP/IP, unlike UDP, is able to provide for a guaranteed data packet delivery, and this is a key requirement for the RTP Audio Application. Control packets are delivered less frequently than the normal RTP data packets and the extra overhead required for a TCP/IP packet is offset by the lower delivery rate.

The underlying protocol must provide multiplexing of the data and control packets. Separate port numbers for the data and control packets ensure this occurs. The RTP Audio Application described in this thesis uses port number 9010 for the data packets and 9015 for control packets.

3.2.2.1. Control Packets

There are five different primary control packets defied for the standard RTCP implementation [Schulzrinne *et. al.* 1996]:

1. **SR** Sender Report, contains transmission and reception statistics from participants that are active senders.
2. **RR** Receiver Report, reception statistics from participants that are not active senders.
3. **SDES** Source description items.
4. **BYE** End of participation.
5. **APP** Application-specific functions.

The only control packet implemented for the test environment of this thesis is the Receiver Report (RR). The difference between the SR and RR packets is that a SR packet includes a 20-byte sender information section used by active senders. An active sender is a user that has transmitted any RT'P data since the last control packet was delivered.

SR and RR packets are not only useful to the server as a means of measuring receiver statistics, but also for other receivers (in a multicasting environment) and also for third party monitoring devices. Ideally, report packets should include a cumulative count, which enables measurements to be performed over both the long and the short term. The QoS Mapper component of the Session Manager performs a comprehensive statistical gathering and monitoring function, thus enabling the receiver to monitor the current network conditions and deliver the results in a control packet. (This is discussed in detail in

Section 3.3.2.2.). Those statistics measured in the test environment that are of relevance to the SR and RR packets include the packet loss rate, average payload data rate, average packet rate and inter-packet arrival time jitter.

Currently, the only participant in the RTP Audio Application that is an active sender is the data server. Transmission of reception statistics from the server is therefore not necessary. However, if the described application is extended to enable bi-directional transfer of data then it will be necessary to implement the SR control packet.

Similarly, a SDES packet is not required, as it is only used to differentiate between control information sent by different users. Delivery of control data that does not fall into any of these predefined packet types is achieved with an application-specific control packet. For example, if the experimental audio application is expanded into a full RTP implementation, then the delivery of the control information advising the server of the new required QoS will be sent as part of an application-specific control packet.

3.2.3. Resource Layer Conclusion

Resource Layer issues are related primarily to topics that are beyond the scope of the experimental system. Networking issues, such as the available bandwidth, packet loss and packet delivery rate, depend on the underlying network structure and cannot be controlled. The adaptive flow system attempts to minimise the disadvantages caused by the unpredictable, underlying network conditions by utilising RTP and RTCP as the transport protocols. These protocols are able to provide ideal information for the adaptive-flow audio application, such as timestamping, sequence numbers and payload type identification. Control packets provide an efficient feedback mechanism, which allows reception reports to be transmitted from the receiver to the sender for the dynamic adjustment of the QoS.

3.3. Application Layer

The Resource Layer is mainly responsible for the primary function of receiving and transmitting data to and from the network. Value-added services, such as renegotiation of the QoS and transfer of the data to the media device, are performed within the Application Layer. QoS issues, including the representation of QoS at the various layers of the system, are also investigated.

The Application Layer has three primary functions: mapping QoS values between the layers of the system; providing functionality for the renegotiation of the QoS, and outputting data at the media service. The QoS mapping functions are described in relation to the mapping between the three system-layers, namely the Resource, Application and User layers. Renegotiating the QoS and outputting the data are implemented as part of a Session Manager, which is responsible for the core components of the Application Layer.

3.3.1. QoS Mapping

QoS mapping deals with the translation of QoS parameters between the User, Application and Resource layers. User QoS requirements are mapped into application-specific parameters for media services, and then into QoS requirements for the underlying resources. User perception of QoS is not yet completely understood, and it is therefore possible to describe a particular QoS representation in a number of different ways for each layer.

QoS representation is not only determined by a particular multimedia data type, but is also dependent on the characteristics of the specific data stream. For example, video data (frame-rate, frame-size, number of colours, colour-depth etc.) exhibits vastly different characteristics to audio data (sample rate, sample size and number of channels).

All QoS mapping occurs through the QoS Mapper component of the Session Manager (section 3.3.2.). The User Layer representation of QoS is in a form that the user is able to easily understand. The maximum deliverable QoS for a particular data type is considered to be 100% QoS. The User Layer mapping of all delivered QoS values is then as a percentage value relative to the maximum value possible.

At the Application Layer, conversion of this user-specified QoS value into a multimedia data-type value is required e.g. 10 frames per second, 300x200 frame size and 8-bit colour-depth. Resource Layer representation is only interested in the bandwidth required for delivery of the specified QoS e.g. 600000 Bps. The QoS Mapper is the core component of the entire system and allows the interactions between the different layers of the system to occur.

3.3.1.1. User level

Many applications currently support only application-level QoS requirements. QoS parameters tend to be difficult for users to understand, and add an extra level of complexity to the system. It is therefore important to specify quality requirements in easy-to-understand user terms.

User requirements of QoS are extremely subjective and difficult to assess quantitatively. An analysis of the user requirements is required in order to determine

a) how the user expects a media service to behave, and
b) how satisfaction of the media service quality can be expressed in quantitative terms.

Although the RTP Audio Application prototype deals with only three QoS parameters (sampling rate, sampling size and number of channels), and one application or resource-specific parameter (bandwidth required), it is important to present a generic mechanism for QoS translation in order for future extensions to be successfully added.

A simple specification mechanism for QoS is required. If QoS requirements are transformed onto a one-dimensional scale, the qualities of the different media become comparable and the scale provides a prioritisation mechanism. A higher available bandwidth implies a higher possible delivered QoS. The RTP Audio Application achieves this scale by transforming the three QoS parameters into a one-dimensional value of the required bandwidth. Samples that require the same bandwidth value are placed in the same QoS group, within which prioritisation can occur.

An experiment was conducted whereby a series of ten 15 second audio samples was played to 25 different users. The audio samples were divided into three groups according to the amount of bandwidth required to deliver the sample. Those samples requiring the same bandwidth were placed in the same group. All groups were separated from each other and users were not required to compare samples of different QoS groups.

Users were asked to sort the samples according to the quality of the audio that they wanted to receive. The worst quality sample was assigned a value of one, and other samples were assigned a value greater by one as they improved in quality. In the case of the user being unable to decide whether one sample was preferable to another, the values were split equally between the samples and they were all then assigned the same value. Figure 5.3 shows the final results of the QoS samples[1]:

[1] Samples were not recorded at 11025 Hz, 8 bit, mono and 44100 I lz, 16 bit, stereo.

Sample Number	Sampling Rate	Sample -size	Channels	Bandwidth	Sum of user values
QoS Group 1					
Sample 1	11025 Hz	8 bit	Stereo	22050 Bps	31.5
Sample 2	11025 Hz	16 bit	Mono	22050 Bps	63
Sample 3	22050 Hz	8 bit	Mono	22050 Bps	55.5
QoS Group 2					
Sample 1	11025 Hz	16 bit	Stereo	44100 Bps	34
Sample 2	22050 Hz	8 bit	Stereo	44100 Bps	54.5
Sample 3	22050 Hz	16 bit	Mono	44100 Bps	78.5
Sample 4	44100 Hz	8 bit	Mono	44100 Bps	83
QoS Group 3					
Sample 1	22050 Hz	16 bit	Stereo	88200 Bps	37
Sample 2	44100 Hz	16 bit	Mono	88200 Bps	61.5
Sample 3	44100 Hz	8 bit	Stereo	88200 Bps	51.5

Figure 3.3. - Audio samples grouped according to QoS groups and user values

These results show that users prefer samples recorded using a 16-bit sample-size and a higher sampling rate than samples recorded in stereo. The priority of the sample within the QoS table is, therefore, determined by the sample-rate and sample-size. If a specific audio stream requires transmission of stereo data, even at low QoS values, the QoS table can be adjusted to cater for this.

3.3.1.2. Application level

Representation of QoS requirements at the Application Layer is the least complicated of the three layers. Although a generic set of QoS requirements for all media services is difficult to define, parameters for only a single media service are reasonably simple to identify. The RTP Audio Application currently delivers only uncompressed audio data. As has been discussed, the sampling parameters referring to the quality of the media stream are the sampling-rate, sampling-size and the number of channels. The period of the media stream is defined as the packet length specified at connection time between the client and server. These parameters are then mapped to resource-level QoS parameters.

Sample Rate	Sample Size	Channels	Bytes/sec
11025	1	1	11025
11025	1	2	22050
11025	2	1	22050
11025	2	2	44100
22050	1	1	22050
22050	1	2	44100
22050	2	1	44100
22050	2	2	88200
44100	1	1	44100
44100	1	2'S	88200
44100	2	1:	88200
	2		176400

Figure 3.4. - Mapping table illustrating the bandwidth values for the RTP Audio Application.

Different media services require different application parameters. For example, video is characterized by temporal and spatial requirements, frame frequency and also colour space. Corresponding attributes for a video stream are the transmission rate, frame jitter, picture resolution, window size, the encoding scheme and the colour depth. Depending on the media service, other parameters that might be introduced include quality, reliability and delay.

3.3.1.3. Resource level

Application-specific parameters are mapped onto resource-level values. Media services that require larger processing power, such as compressed video data, require resources such as the operating system, CPU type and load and thread scheduling to also be investigated. Due to the nature of the uncompressed audio data delivered in terms of this thesis, the only resource in the RTP Audio Application that requires examining is the network. The application level QoS parameters are mapped directly to a bandwidth value. This value determines the network resources required for processing the media stream.

System Layer	Mapping	Values
User Layer	Percentage of Maximum QoS	0 — 100 %
Application Layer	User Layer value mapped to a value between the minimum and maximum QoS. The percentage value cannot be mapped onto an exact value but after performing a lookup into the QoS Table, the closest QoS value is used	**0** -> (11025, 1, 1)
		50 % -> (44100, 2, 1)
		100 -> (44100, 2, 2)
Resource Layer	Application layer QoS values are mapped to a quantitative Resource Layer value. Bandwidth = sampling rate * sample size * number of channels	(11025, 1, 1) -> 11025
		(22050, 2, 1) -> 44100
		(44100, 2, 2) -> 176400

Figure 3.5. - QoS Mapping between system layers

Figure 3.5 provides a summary of the system layers and the mapping functions for translating QoS values between different layers.

3.3.2. Session Manager

The Application Layer is responsible for most of the multimedia data processing and also the QoS Mapping functions. In order to improve the flexibility of the components within the Application Layer, the separate components are grouped together to form the Session Manager. Alfano and Radouniklis originally proposed the Session Manager as a component of The CME[1] [1996]. As can be seen in Figure 3.6, the four components of the Session Manager are the Connection Manager, QoS Mapper/Controller, Resource Monitor/Controller and the Service Manager. The Media Services component is not part of the Session Manager, but interaction between the Service Manager and Session Manager does occur.

[1] CME — Co-operative Multimedia Environment

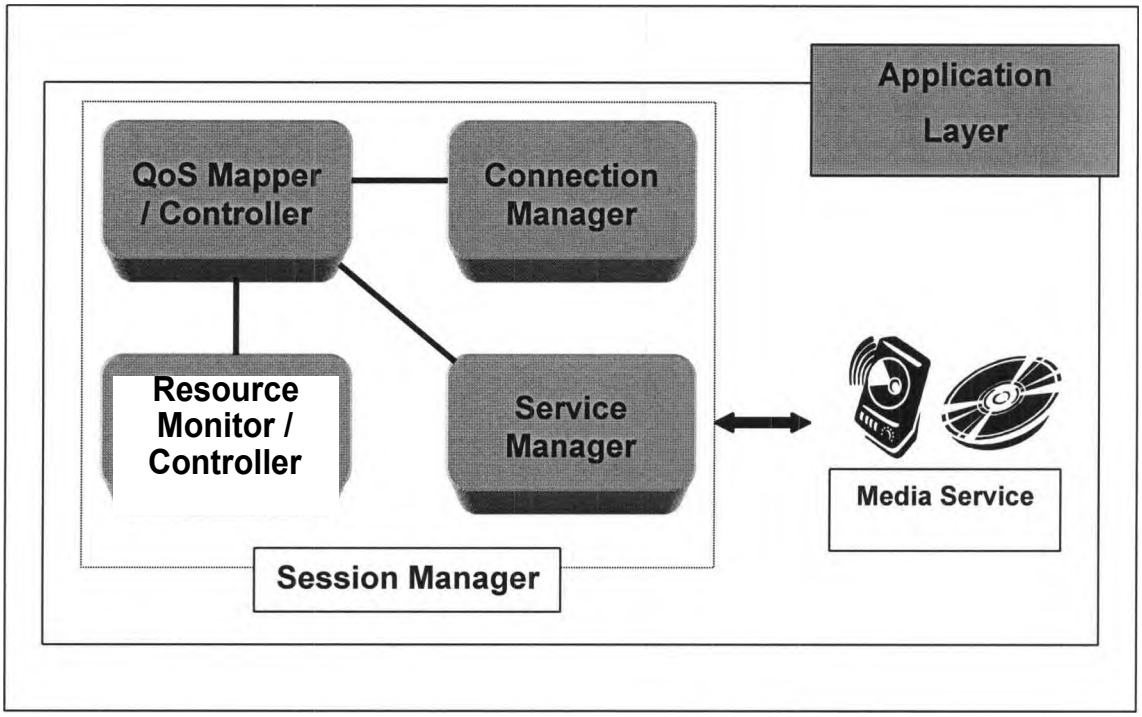

Figure 3.6. - The Application Layer Session Manager

3.3.2.1. Connection Manager

The Connection Manager is responsible for the establishment and termination of a session between participants. The CME Connection Manager is based on a distributed peer-to-peer model, and therefore does not rely on a centralised session moderator. The RTP Audio Application is based on a client-server model; thus a complicated session establishment and termination procedure is not required. The client informs the server that it would like to establish a connection. After consulting with the Resource Monitor / Controller to ensure there are sufficient resources to accept the connection, the Connection Manager either honours or dishonours the request accordingly. If the connection is accepted, the Resource Monitor / Controller is informed of the new connection. In the RTP Audio Application, this procedure is physically administered within the RTP Library.

RTCP connection requests are managed in the same method. A RTCP connection can only be created after the RTP connection has been successfully completed. The RTP connection request contains the IP address of the client requiring the connection, which is necessary for the RTCP connection to be completed.

3.3.2.2. Resource Monitor / Controller

According to Alfano and Radouniklis, four QoS scenarios can be outlined:

1. Host and network resources both provide QoS.
2. Host and network resources do not provide QoS.
3. Only host provides QoS.
4. Only network provides QoS.

The resources that are monitored by the Resource Monitor are the network (measured in kBps) and the host resources (CPU, operating system, media services etc.). Host resources are only controlled in a system where QoS guarantees are provided across the entire end-to-end structure. QoS Architectures have been developed specifically with the goal of providing end-to-end QoS guarantees. The RTP Audio Application does not consider host resource QoS guarantees and therefore the Resource Monitor reviews only network resources.

If the network or host resources do provide QoS guarantees (situations 1,3,4 above), then the Resource Monitor / Controller performs three actions:

1. Reserve and allocate resources during the session establishment stage to allow traffic to flow according to the QoS specification.
2. Provide resources during the lifetime of the connection according to QoS specification.
3. Adapt to resource changes during multimedia data processing.

In the situation where no QoS guarantees are provided by the host or network as in the RTP Audio Application, the Resource Monitor / Controller is responsible for checking the status of all system resources and providing the QoS Mapper with the monitored resource information. The system is designed to operate across heterogeneous IP networks with no QoS guarantees. The Resource Monitor is therefore only responsible for monitoring data received from the network, and for determining the network resource usage. Physically, the Resource Monitor resides within the QoS Mapper component in the RIP Library. This design decision was made in an attempt to reduce system overheads and increase efficiency.

Assuming the establishment of a successful connection from receiver to sender, the initial QoS to be delivered is calculated by the Resource Monitor. The initial bandwidth value can be determined in a number of different methods. A series of RTP data packets delivered from the sender, and then monitored by the receiver, will provide a current throughput value, or a packet loss ratio. Depending on the renegotiation method that is currently implemented, the initial QoS is determined and a RTCP packet is returned to the server containing the initial QoS value. This method proved problematic with

the implementation of the RTP Audio Application, in that the session establishment is reasonably lengthy, and this initial negotiation would increase the time period required. Although not as accurate, the method currently in place is that the receiver requests an initial QoS value (depending on whether a high or low starting QoS is required).

In addition to calculating the initial value of the necessary bandwidth, the Resource Monitor is also responsible for monitoring the network resource throughout the lifetime of the connection. Whilst monitoring the network resource, the Resource Monitor also calculates various statistics which are then used for the renegotiation of the delivered QoS. The renegotiation formulae that have been implemented and their corresponding results are discussed in Chapters 5 and 6.

Monitoring of the network resource results in a new available bandwidth being calculated. This bandwidth value is passed to the QoS Mapper in order to determine the new QoS that is to be delivered.

3.3.2.3. QoS Mapper / Controller

The QoS Mapper / Controller is responsible for translating user requirements into application-specific QoS values and also for translating media-specific parameters into QoS requirements for the underlying resources. The QoS Mapper requires each media service to define a mapping table that contains all the possible QoS parameters and the resulting bandwidth that will be required for each combination of parameters. The mapping table therefore forms the core component of the QoS Mapper.

Within the Session Manager, the QoS Mapper interacts with both the Resource Monitor and the Service Manager. If a renegotiation of bandwidth is required, the Resource Monitor passes the QoS Mapper the available bandwidth value. The QoS Mapper uses this value as a lookup into the mapping table, and the table returns a media-specific QoS parameter. The media-specific parameter is passed to the Service Manager and then the delivered QoS value from the media service is altered.

The mapping table for the Audio Application is illustrated in figure 3.4. The Resource Monitor will, for example, pass a value for the available bandwidth of 50 kBps to the server. The QoS Mapper then determines that eight possible QoS values are able to be delivered with the available resources. The Service Manager adjusts the sampling parameters of the audio device and the QoS is then automatically adjusted. Following this, the data at the new QoS, is delivered to the client.

3.3.2.4. Service Manager

The Service Manager provides the mechanisms to stop and start media services. Retrieval and control of media-specific parameters for the dynamic adaptation and reconfiguration purposes are also duties of the Service Manager. A customised interface to lower level controls of the various media services is enabled via this mechanism. This enables specific parameters for media services to be easily altered and customised. The Service Manager is used either by the QoS Mapper or directly by the User Layer. The user interface employs the Service Manager's functionality to start, stop and adjust media services. The QoS Mapper, however, retrieves and adjusts media-specific parameters via the Service Manager.

3.3.3. Media Services

As already discussed, the Media Service is not a core component of the Session Manager. Media services are highly dependent on the multimedia device currently in use, and a generic mechanism to control all available media services is not possible. Separation of the media services from the Session Manager is necessary to enable each application the mechanisms to implement the required device.

Control of the media service QoS is achieved via parameters passed from the Service Manager. Specific device controls, such as play, stop etc., are controlled by the media service interface and therefore knowledge of the underlying device is not required. Each media service requires a separate interface specific to the device type.

The RTP Audio Application currently outputs only uncompressed audio data. The system is, however, designed to output all multimedia data types. Current design restrictions require the media service to process three QoS parameters, the sampling rate, sample size and number of channels. These parameters are discussed in detail in section 2.4.2.

Design constraints are required in order to construct the QoS mapping table (discussed in the Application Layer). An upper and lower bound is placed on all QoS parameters. The sampling rate is restricted to the values of 11025 kHz, 22050 kHz and 44100 kHz. The maximum rate of 44100 kHz was chosen, as it corresponds to the maximum sampling rate of an audio CD player. Three different sampling rates are specified in order to keep the QoS mapping table to a reasonable size. There is, however, no design limitation on the number of sampling rates that are implemented.

Stereo and mono audio streams are possible, surround sound and other multi-channel systems are, however, not possible. The number of channels is limited to 2. Similarly, sample size is restricted to

either 8- or 16-bit sound quality. A number of different audio samples are possible, including compressed audio format. The RTP Audio Application is designed primarily as a prototype system for the adaptive flow of multimedia data, and numerous audio formats are not required for this.

3.3.4. Application Layer Conclusion

QoS issues have end-to-end scope and encompass the Resource, Application and User layers. The delivery of an adjustable QoS flow, with constant information content, requires delivery of a constant QoS for each renegotiation period. Although adaptive flow systems do not require a guaranteed QoS through all system layers, the relevant issues required to deliver a constant QoS are nevertheless important for the test system proposed in this work. QoS mapping between layers of the system is a core component of the RTP Audio Application, and an accurate representation of the QoS at each system layer is required. QoS representation at the User Layer is intended to be simple and intuitive for the end-user, and enables an abstraction from the lower-level technical QoS aspects to be made.

The component nature of the Session Manager in the Application Layer allows for a separation of these distinct, but related, tasks. Mechanisms required for the dynamic renegotiation of the QoS are provided by the Resource Monitor, the QoS Mapper is responsible for maintaining a consistent and accurate representation of the delivered QoS, and the Service Manager allows for interactions to occur between the Media Service and the Application Layer. The test system discussed throughout this work uses the Open Sound System to dynamically adjust the QoS of the delivered audio stream.

3.4. User Layer

The User Layer is the only part of the system that the user interacts with and is therefore the only part of the system that most users require any knowledge of. Currently, the graphical user interface is designed to be reasonably simplistic. The user is able to enter the remote server's IP address and also the port to connect to. After the connection has been established, it is possible to end the connection at any time.

The text-based interface is far more powerful, although more difficult to use. Any of the possible variables (such as sampling rate, number of channels, packet interval, packet size etc) can be set at connection time. The user can either define the QoS at connection time, or the Resource Monitor sets the initial value. This initial value will, however, alter throughout the lifetime of the connection.

3.5. Concluding Remarks on System Design

A hybrid system is proposed that combine a layered approach to the design of the entire system, as well as developing the core components of the Application Layer as part of a Session Manager. Existing systems do not allow for the simple delivery of variable QoS data. QoS-Architectures attempt to provide an entire end-to-end system for the delivery of a constant, guaranteed QoS, but these systems are not possible to implement over heterogeneous IP networks, such as the Internet, due to the complex non-predictable nature of the current generation of IP networks. A system is therefore proposed, using existing, widespread networking technologies, that provides for the delivery of a constant information content, with a dynamically adjusting delivered QoS. Implementation of the system is required only in the application end-system, and a layered, component-based approach to the problem is proposed.

Separation of the User, Application and Resource layers allows distinct, but related tasks to be completed by the specific layer. The Resource Layer protocols utilised in the described RTP Audio Application provide mechanisms ideal for the real-time nature of data delivery required in the application. The Application Layer provides the core components required for the renegotiation of the QoS. Interaction between different system-layers is achieved by mapping the QoS into a simple and constant, but accurate, representation of the delivered QoS.

Justification of the system design principles is provided in the following chapters. Implementation issues arising during the construction of the RTP Audio Application, designed in this chapter, are discussed in the next chapter, which includes a detailed description of the underlying structures as well as networking and application issues, such as synchronisation and buffering.

Chapter 4

Implementation Issues

4.1. Introduction

Having reviewed the design of the proposed RTP Audio Application, it is necessary to implement the system in order to prove that the adaptive flow system is possible, and to test the effectiveness of the renegotiation methods proposed in Chapter 5. The implementation concentrates on three primary design areas, as well as issues relating to these areas and the system as a whole.

The Resource Layer functions are put into effect using an existing RTP implementation. This chapter examines the physical layout of the system in relation to the system design, as well as the overall aims of providing an adaptive flow system for the delivered QoS. The RTP Audio Application requires a dynamic adjustment of the recorded data stream, and the mechanisms that enable this operation are also discussed. Coupled with this, synchronisation and buffering issues within the media device and application have frequently arisen. Synchronisation issues are not only restricted to synchronisation between the media devices and application, but also synchronisation of data packets across the network.

Implementation of the renegotiation methods that provide for the dynamic adjustment of delivered QoS are discussed separately in Chapter 5.

4.2. Networking Issues

The RTP functions are implemented using an RTP Library. The RTP Library facilitates the RTPD', RTCP, RTP API, and a set of message queues allowing data to be passed between the application and the RTPD.

Figure 4.1. illustrates the physical implementation of the RTP Audio Application. The RTP daemon is completely separate from the client program. The RTPD is responsible for the effectuation of the RTP protocol, as well as the statistical functions, whereas the client program is responsible both for interfacing with the device that the data will be acquired from and then for passing the data to the

---

RTPD — RTP dacmon

RTPD for delivery. The client also requires facilitation for receiving data from the RTPD and outputting it to the appropriate media device.

Interaction between the RTPD and client application is achieved via the RTP session class. The RTP session class is responsible for executing the RTP daemon and creating the message queues that allow the daemon to communicate with the application. All data passed between these programs is done via a shared message queue system.

4.2.1. **RTP** Library

The RTP Library consists of an RTP daemon and a programming API. The API allows applications to access the RTP daemon in order to transmit or receive data. Implementation of this version of the library has been completed using C++ by the Institute of the Italian National Research Council [CNR]. The version of the library that was used in the RTP Audio Application was a beta version (0.74) and was therefore not complete. In fact, it is warned that the RTP Library is not tailored for efficiency or robustness, but has been designed solely for research purposes. As previously discussed, the Application Layer does not implement RTCP nor have any support for multicasting.

Although RTCP support has been added for the RTP Audio Application, multicasting is not required. Problems experienced during the implementation and effectuation stages for RTCP are discussed in section 4.3.

The entire RTP system can be broken down into four components (shown in figure 4.1):

1. The RTP daemon
2. RTP message queues
3. RTP API
4. Application

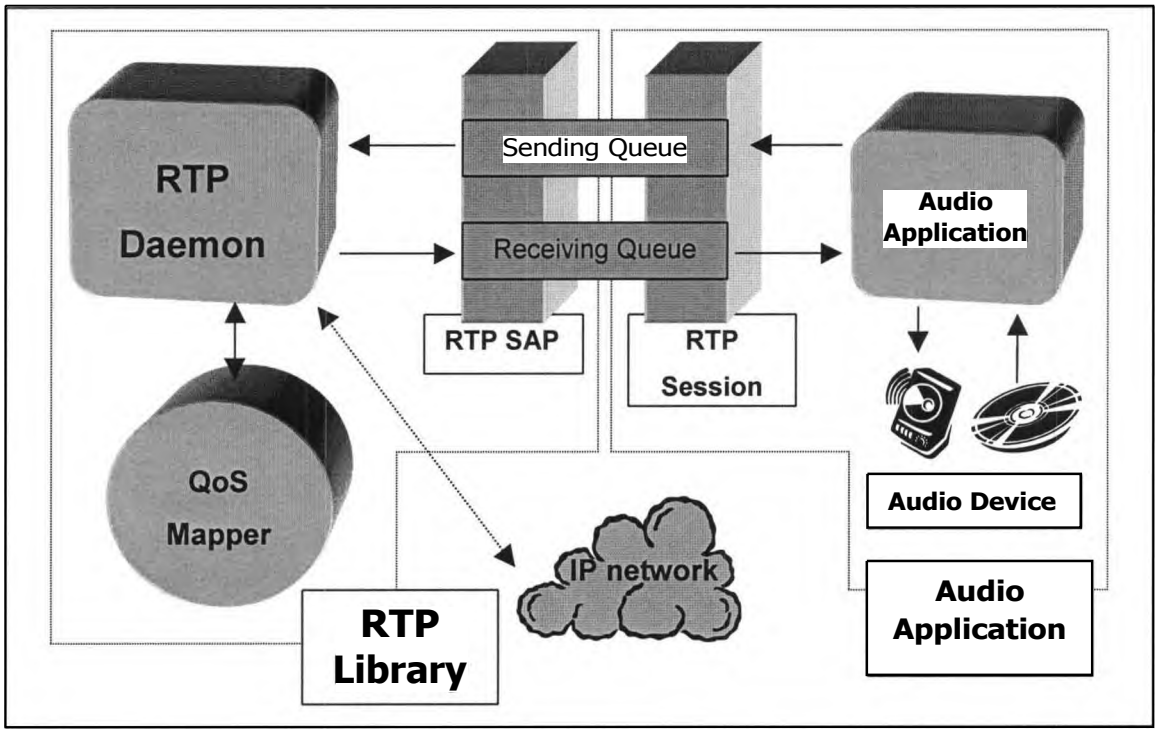

Figure 4.1. - Physical Layout of RTP Audio Application including data flows

4.2.1.1. **RTP Daemon**

This is the portion of the system that is the effective consummation of RTP and RTCP. The RTP daemon handles all networking aspects of the system, as well as interacting with the timers and other miscellaneous services.

a. Memory Management

Memory management functions are also abstracted from the underlying system calls. Different operating systems and machine architectures access system memory using different methods. It is therefore possible to use many different memory access methods within a single class. Depending on the selected operating system (e.g.: #IFDEF _LINUX_) the system call is implemented accordingly. Other class libraries within the Application Layer are then able to use the memory method regardless of operating system or platform.

b. Signals

A signal is a software interrupt that causes a process to stop executing and perform one of a number of different actions instead [Tranter 1996]. When a program is executing, many actions may occur which are not controlled by the program. These events cause a signal to be sent to the process. The default action for receiving most signals is to terminate the process. It is possible, however, to change the default action of a signal and replace it with the code that is to be executed if a specific signal is received. This is known as a signal handler.

Levels of abstraction occur within the signal handling classes as well. System calls are implemented in a separate class and are accessed by a higher-level library, which can then be utilised by other classes in the RTP Library. The higher-level class allows blocking and unblocking of certain signals. This is necessary if a signal will cause a function to fail if the signal is received whilst the function is being executed. A signal handler can be declared that describes a signal and the code to be executed if the particular signal is caught. A list of signal handlers can be created that is able to catch a number of different signals. Each signal will also have a corresponding handler attached. For example, the SIGTERM signal can be caught. Instead of simply exiting the program, it is possible for the program to be executed cleanly any removing any remaining data structures. Any data not yet written to a file will not be lost, and if it is a network application, then it is possible to send a data packet indicating that the connection will soon be terminated.

c. Buffering

Depending on the rate that data is received from the network, some form of buffering may be needed to store the data before it is processed. Although the application is responsible for processing and outputting the multimedia data, the RTP daemon needs to collect all the required statistical information. The processing of the data is as follows: the data is received by the network; all IP, UDP and RTP headers are then removed from the packet; the data is then either stored in a buffer or analyzed by the statistics component. The remaining multimedia data can then be transferred to the application via the RTP SAP. The memory management functions provided by the RTP Library are used for storage of the RTP data buffers. Monitoring and control of the buffers is achieved by utilising the list-processing functions of the library.

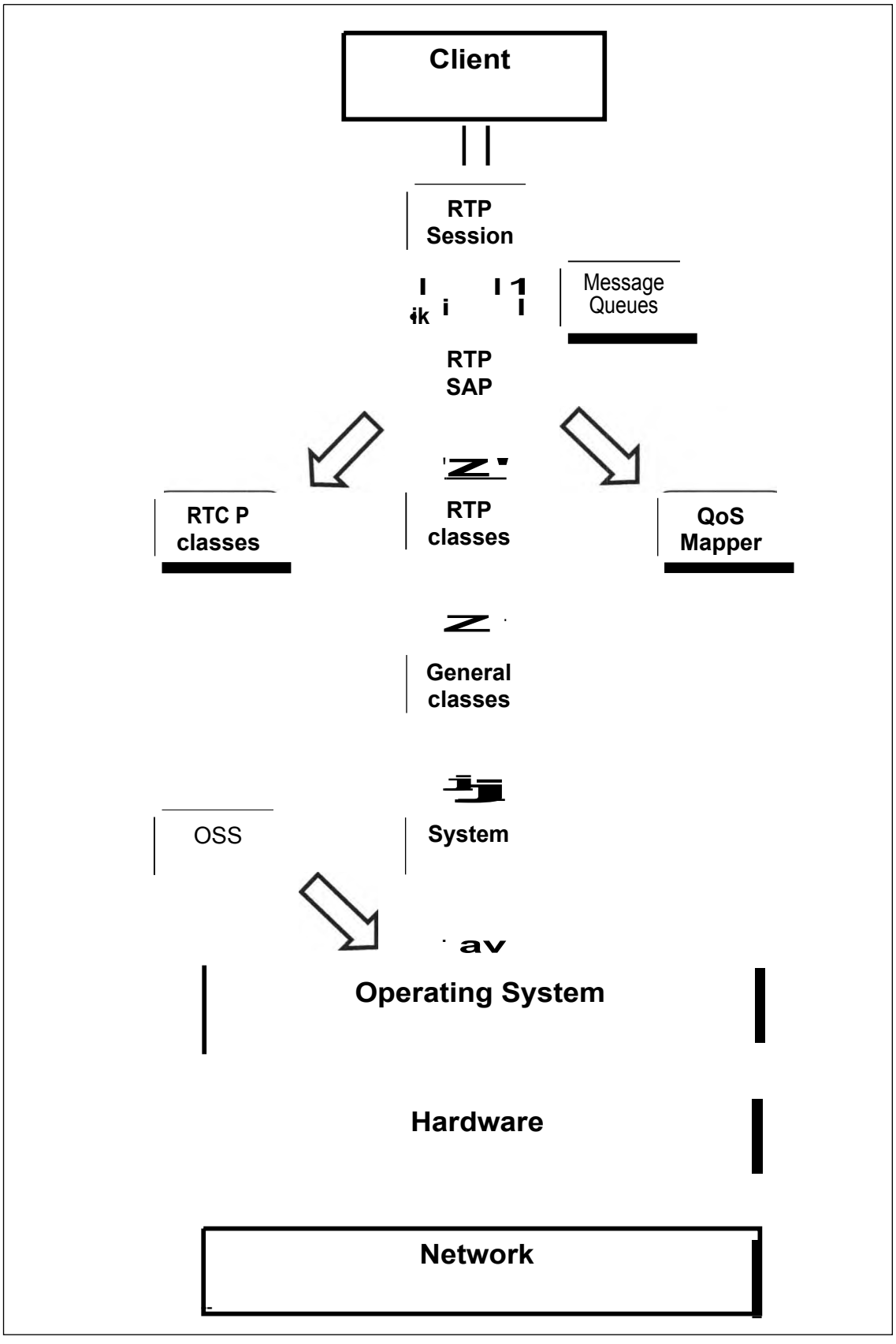

Figure 4.2. - RTP Library hierarchy of calls

d. Data Reception and Transmission

All RTP and RTCP data is received or transmitted via the Resource Layer. In addition to this function, other networking activities such as opening and closing an RTP session, adding new session members, error-checking packets, stripping the packet headers and also determining the origin of the packet are accomplished by the RTP daemon. Specific functions designed for the integration of MPEG audio and video within the RTP daemon have also been included in the library. Due to the RTP Library still being completed, these functions have not yet been completed, and will therefore not be discussed.

e. Statistical Functions

In order for the adaptive flow method of renegotiating the bandwidth to be successful, complete and accurate statistics must be maintained for the entire lifetime of the connection. The Resource Monitor component of the session manager is responsible for the renegotiation process. For every packet that is received, the sent timestamp (timestamp from the server), packet sequence number and data packet size is passed to the QoS Mapper. A received timestamp for the data receiver is also calculated.

All information is then stored in a data table, which enables both the current and cumulative statistics to be calculated. Whether or not the cumulative or current statistics are used, depends on the renegotiation method currently in place. The collected statistics are then output to a file to allow the data to be graphed.

4.2.1.2. RTP SAP

The RTP SAP is a set of message queues that enables data to pass between the RTP daemon and the client application. The RTP daemon is intended as a general purpose **RTP** implementation, providing functions for any client application to use. In order to completely separate the application from the daemon, it is necessary to have some method of transferring data between the two processes. Shared system queues provide this function. The daemon can therefore be executed independently of the application. Although the application requires the daemon for transmitting and receiving data, the daemon allows the RTP functions to be decoupled from the Application Layer and transferred to the transport layer. Most RTP applications implement the RTP data sending and receiving as part of the Application Layer rather than the Transport layer [Schulzrinne *et. al.* 1996]. By decoupling the RTP functions from the Application Layer, system design and implementation is simplified.

Two RTP SAP message queues are created; one for receiving messages and the other for sending messages. The application interacts with the daemon via these message queues. If a data packet is received from the network, the data is sent across the queue as a data message, where it is received by the application.

The RTP session API also creates two message queues. These queues are linked in memory with the RTP SAP queue. Data that is placed on the message queue by the RTP session (in the case of sending data) can be automatically passed to the RTP daemon due to this linked message system. Data is received in the same method.

Message queues are constructed using the Application Layer queue and message classes. Abstraction of the underlying system calls is implemented in the same manner as the other classes. Although not currently implemented, a priority class is available and can be used for prioritising messages between the application and daemon.

4.2.1.3. RTP Session

The RTP session is the API that allows the application access to the RTP daemon. As described in the previous section, data is passed between the daemon and RTP session by means of a linked system queue. The RTP session facilitates the opening and closing of an RTP and RTCP session (including adding a new member to a multicast session), transmitting and receiving RTP and RTCP data, and also checking the current QoS value. If there are any additions to the RTP daemon to be used by the application, message-passing mechanisms must be implemented in the RTP SAP and RTP session and the API also needs to be extended to provide this.

4.2.1.4. RTP Application

The application is the part of the system that the end-user interacts with, and is responsible for outputting the multimedia data as well as interfacing with the RTP API to receive and transmit data. Logically, sections of the RTP daemon (such as the statistics manager) are part of the Application Layer but physically they are situated within the RTP daemon. They are therefore discussed in detail in the Application Layer, rather than the Resource Layer. Most of the assistance provided by the RTP session is used by the application in order to create both an RTP and RTCP end-to-end session. The RTP Audio Application is also responsible for altering the desired QoS parameters and recording data at the new parameters.

4.2.1.5. RTCP

Originally, RTCP support was not available in the RTP Library. Renegotiation of the QoS requires the transmission of control information from the client back to the server. Sampling parameters are altered on the audio device according to this new information. The RTP implementation does not allow for a bi-directional flow of packets, and it is therefore not possible to transmit RTP packets from the client back to the server. An RTCP implementation is therefore necessary to assist the transmission of the control packets.

The current RTP implementation uses UDP as the underlying transport protocol. However, control information requires guaranteed delivery, which UDP is unable to provide. Modifying the RTP Library to deliver all data using TCP/IP as the transport protocol would have affected the nature of the non-guaranteed delivery of the data packets, and was therefore also not an option. As a result, the RTCP implementation delivers only QoS control information. Control packets that are required for other RTCP functions require further effectuation.

The RTCP functions that are utilised within the RTP Audio Application are identical on both the receiver and server. Currently, the transfer of control information is purely from the data receiver to the data transmitter. A complete listing of the RTCP code discussed in this section can be found in Appendix 1.

4.2.2. RTP Library Conclusion

The Resource Layer functions are implemented using an existing RTP Library. The RTP Library is not, however, restricted to only Resource Layer functions. Networking facilitation is provided primarily by the RTP daemon. Establishment of data connections, and the reception and transmission of data packets, are the key RTP daemon functions. In order for these key functions to be successfully employed, the RTP daemon provides secondary functions such as memory management, data buffering and event and signal handling.

Message queues provide message and data passing functions between the main application and the RTP daemon. These message queues allow for the separation of the Application and Resource layers discussed in Chapter 3. Delivery of the multimedia data stream from the Resource Layer to the Application Layer is also achieved via this mechanism. Logically, the Application Layer components are organised separately in the Session Manager. Physically, however, the media services and the Service

Manager component are implemented as part of the RTP Audio Application. The other Session Manager components are implemented within the RTP daemon.

The addition of RTCP support to the existing RTP Library structure was required in order to deliver control information from the client to server. Because control information requires a guaranteed delivery mechanism, TCP/IP is used as the underlying protocol rather than UDP.

4.3. Audio Application

4.3.1. Introduction

The RTP daemon is responsible for transmission of data between the server and client, and the message-passing facilities of the RTP SAP component enable the transfer of data between the Application and Resource layers of the system. Transfer of data between the media device and Session Manager occurs via the Service Manager component. The multimedia data flow, therefore, follows this sequence: media device, Service Manager (Session Manager in the Application Layer), RTP SAP, RTP daemon (Resource Layer), networking component, and is finally delivered to the client. The same flow, in reverse, is then followed at the receiver.

Careful synchronisation between the media device and Service Manager, as well as between the server and client is therefore necessary. Synchronisation issues, as well as the buffering required at various system layers, are discussed later in this chapter.

The default settings for a sound recording device under OSS are unspecified, thus making it necessary to either inform the user of this, or to include a mixer program in the application [4Front]. Mixers are not present on all sound cards, and including mixer features will make the application more hardware dependent. Similarly, it is necessary to set all sampling parameters that the application depends on. It cannot be assumed that all hardware devices will support all sampling parameters. If an application relies on the use of stereo, then the device must explicitly set the number of channels, and an error message returned if it is not supported.

4.3.2. Buffering

Audio data requires that samples are played continually with no breaks in the delivered service because lost and late packets greatly affect the quality of the received audio stream. The RTP Library uses UDP as the underlying transport protocol, and therefore no guarantee can be made of timely packet delivery. The RTP Audio Application attempts to reduce the number of late and lost packets delivered to the client by introducing buffering within the system to reduce the breaks between the samples that have been delivered on time.

Applications using sound devices need to declare buffers in order to store the information that has been read/written from the device. The size of the buffer determines the amount of data that will be delivered every time the device is accessed. A large buffer reduces the amount of system call overhead, but a small buffer can improve real-time performance [4Front]. The application does not access the audio hardware directly. Data is passed from the application to the kernel DMA buffer and the audio device is able to access it from here.

Traditionally, a double-buffering method has been used in audio applications. Two buffers are available to the application. Whilst one is being written to by the application, the other can be accessed by the audio device. After processing the first buffer, the device then begins to process the second buffer. This action is repeated until the device is closed. The application is therefore able to process information whilst the device processes audio data. Using this method, recording and playback of an audio stream without pauses is possible [Tranter 1996].

Processing low-quality audio data using the double-buffering method is reasonably effective. If an application is recording data at 8kHz/8-bit/mono and there is 8kB of data in the buffer, the application has 0.5 seconds to read the data from the buffer, store it to disk and then return to read from the device again. Problems begin to occur when high quality data, such as CD data (44100 Hz/16 bit/stereo), is transferred. At a transfer rate of 176 kBps the application has 23 milliseconds to process the information in the buffer. Processing times can be improved by utilising larger buffers, this however, increases the required processing overhead [4Front].

OSS employs a buffering system known as multi-buffering. The available buffer space is divided into equal size blocks known as fragments. According to the OSS programming documentation, multi-buffering increases available buffer size without increasing the overhead latencies [4Front]. The buffer size depends on the sampling parameters (rate, channels, sample size) that the application sets.

The audio device will only begin to read data from the buffer when an entire fragment is filled. In non-blocking mode, control is returned to the application and it is then able to fill the remaining buffer fragments. If the application attempts to write more data than there is available in the buffer, the application is put in blocking mode, which means that the application must wait until space is available in the buffer before it is able to process more information. All processing by the application is therefore halted and data delivered across a network or received from an input device may be lost.

The opposite problem may occur if the audio device is outputting data faster than the application can process it. This situation occurs if the CPU is too slow or many other devices are using the processor. This playback-underrun situation also occurs when there is no new data written to the audio buffer before all data currently in the buffer is output, which results in noticeable delays occurring between audio samples. Similarly, a recording overrun situation occurs when the audio device fills the recording buffer completely. The device is then stopped and further samples being recorded will be lost [4Front].

4.3.2.1. Determining Buffering Parameters

Determining buffer sizes whilst using a traditional double-buffering mechanism is generally related solely to performance issues. As mentioned previously, large buffers reduce the system overhead required, but a small buffer size improves real-time performance. The OSS model determines the optimum fragment size by measuring the sampling parameters and the available memory. This fragment size is calculated as the ideal trade-off between performance and overhead.

a. Static Buffer Information

The static information relating to the buffer and fragment size can be determined by using the following i o ct 1 ( ) call:

```
ioctl(audio device, SNDCTL DSP GETBLKSIZE , &frag size)
```

The fragment size is returned in `fra g_s ize`, which is then used to determine the required size for the application buffer, as well as determining the number of bytes to be read from or written to the device.

Using an incorrect fragment size may have unforeseen results on the audio output. Not only can synchronisation errors between the application and the audio device occur, but attempting to output samples that are too short results in noise. It is therefore inadvisable to reduce the buffer size in order to improve the real-time performance of the system.

b. Dynamic Buffer Information

The current buffer status can also be retrieved. This returns more information than just the fragment size.

ioctl (audio device, SNDCTL_DSP_GETISPACE, &audio_buf_info) returns the input buffer information. Similarly, ioctl(audio_device, SNDCTL _ DSP _GETOSPACE, &audio_buf_info) returns the output buffer information.

The audio _ buf _info data structure contains the following fields:

1. fragments is the number of full fragments that can be read or written without blocking occurring.
2. fragstotal returns the number of fragments allocated to buffering.
3. fragsize is the size of the fragment in bytes.
4. bytes returns the total number of bytes that can be read or written without blocking.

4.3.2.2. Buffering Strategy

Implementation of a buffering system that only read or wrote full fragments to or from the audio device caused audible breaks in the audio stream. After each sample was output, a small, but audible, break in the music could be heard. The length of the sample determined the frequency of the breaks. The longer the sample, the less frequently the breaks occurred. In order to prevent the breaks in the audio stream from occurring, a series of secondary buffers was also implemented.

The secondary buffers consisted of a MAX_NUMBER of buffers, each of size 65535 (the maximum packet size for a UDP packet). The aim of the secondary buffers was to store the audio data into large segments and then deliver entire fragments to the audio device. A reduction in processing time for each buffer fragment would therefore occur, and it was believed that this would prevent breaks in the audio stream.

The secondary buffering system did not seem to alleviate the problem, and in fact new problems, such as the breaks between samples increasing in length, were introduced. Adding extra buffers was, therefore, not the solution to the problem and as a result, the extra buffers were removed from the application. Another attempt to solve the problem of the breaks in the data stream is discussed in the Synchronisation Issues section (4.4.3).

4.3.3. Synchronisation Issues

In order to prevent synchronisation errors, full samples must be extracted when reading from or writing to the sound device. If this does not occur, then either noise will be output, or the left/right channels will be swapped around. A full sample can be calculated as the number of channels * number of bits used for encoding. For example a 16 bit, stereo sample will be 4 bytes and an 8 bit, mono sample will only be 1 byte.

4.3.3.1. Application – Sound Device Synchronisation

Synchronisation of data between the media service and the service manager caused numerous problems. Many of the problems related to buffering (discussed in the previous section) are also associated with the synchronisation problems.

Implementation of a larger number of buffers, or increasing the size of the existing buffers, did not appear to solve the delay problems experienced when transferring data from a buffer to the audio device. No delay problems were experienced if the entire file was read into one large buffer, which was then written to the audio device. The device did block, however, until all data was written to it if the file was larger than the available buffer size. If the input data was separated into two or more buffers and transferred directly to the device, a break in the audio stream was clearly audible between each sample.

Using buffers to prevent these breaks in the audio stream proved unsuccessful. The fact that the data did not break up if delivered as a single sample indicated that the complication existed in the code that was processed before each sample was output to the sound device. Sampling parameters for every sample are written to the sound device before the sample is output. This ensures consecutive samples recorded at different sampling rates can be output without the device having to be closed and then reopened and the new sampling parameters set. As was discussed in section 4.1.2., sampling parameters may only be set between the device being opened and the first reading from, or writing to the device, or if an ioct 1 ( ) call was made. The i oct ( ) calls that are used to allow the device to accept new parameters are SNDCTL DSP RESET and SNDCTL DS P SYNC.

The major difference between the two control parameters is the way in which sampling parameters are handled. SNDCTL DS P SYNC delays returning control to the application until all bytes currently in the buffer have been output to the audio device. Using SNDCTL DS P RESET causes control to be returned to the application immediately, and all data in the buffer is then lost.

Originally, SNDCTL_DSP_RESET was implemented in the audio code. The sampling parameters are altered whenever a new sample is transmitted to the audio device and this caused all data currently in the audio buffer to be lost. Changing SNDCTL_DSP_RESET to SNDCTL DSP_SYNC fixed the small, audible break in the audio stream successfully.

New complications arose as a result of this change. Consecutive audio samples sampled using the same sampling parameters and then transferred to the audio device were output properly. However, samples recorded using different parameters did not work correctly. Although the OSS documentation indicates that SNDCTL_DSP_SYNC can be used to change sampling parameters, this clearly was not the case. Any sampling parameters that were altered after calling SNDCTL_DSP_SYNC were not implemented.

It was therefore necessary to revert to the original SNDCTL_DSP_RESET call. Audio data already present in the buffer was lost every time the call was implemented. Changes in sampling parameters were not implemented when the call is not used. Solving this problem consisted of checking the sampling parameters for each new sample, comparing the parameters to the previous sample, and if there was any difference an ioctl ( ) call using SNDCTL_DSP_RESET had to be made. This had the unfortunate side effect that all data currently in the data buffer at the old sampling rate was deleted when data at a new sampling rate was received. In the RTP Audio Application, sampling parameters are only adjusted when a renegotiation of the QoS occurs. The frequency of the renegotiations therefore determines the amount of audio data lost due to this inconsistency in the OSS implementation.

Implementing the current sampling parameters only when there are any differences in the sampling parameters also creates synchronisation problems. Initially, it was attempted to place the ioctl ( ) and the changing of sampling parameters into a separate class method. Again it was discovered that this method did not work. The new sampling parameters were never implemented, and all data was output at the original rate. The ioctl ( ) call and the sampling parameters have to be implemented directly after each other. No other code can be executed between calling the sound control code and setting the new device parameters. This problem was solved by placing the SNDCTL_DSP_RESET call directly before the data is read from, or written to, the device.

4.3.3.2. Sender — Receiver Synchronisation

Synchronisation of audio data between the audio device and the application requires the use of buffers and sound control calls. Data transferred across a network from a sender to a receiver also requires synchronising. Data sampled using a particular set of sampling parameters must be output at the receiver using the same parameters. If this does not occur, then the user receiving the data will hear noise.

a. Dynamic Fragment Adjustment

In order to prevent buffering problems, the OSS multi-fragment buffering method requires that each sample is delivered to the audio device as a full fragment. In a sender-receiver situation this requires that both the client and the server read and write the same size fragments to the audio device. If all audio data delivered from the sender to the client is sampled using the same sampling parameters then no problems should occur.

The RTP Audio Application does, however, pose a few problems. The audio data is not always sampled using the same parameters. Fragment size depends on the sampling parameters and the amount of memory available. This means that every time the QoS is altered, the buffer fragment size and therefore the amount of data that should be read from or written to from the audio device, also changes. Attempting a dynamic readjustment of the application audio buffer for each QoS renegotiation requires a large processing overhead and is therefore not implemented.

Due to these factors, it was decided that the fragment size would be set for the entire connection, and is determined by the initial sampling parameters. Data not completely filling a fragment is lost if any renegotiation occurs, due to the use of the SNDCTL DSP_RESET call when the sampling parameters are changed.

b. Sampling Parameters Synchronisation

The delivered QoS of the audio stream is determined by the renegotiation method currently in place. Only the receiver monitors the statistics that determine whether to alter the QoS. If any changes need to be implemented, the client transmits an RTCP packet to the sender indicating the new QoS to be delivered. The sender alters the sampling parameters to reflect the new QoS, and data is then delivered at this new rate.

Originally, after the receiver determined that a renegotiation of the QoS was necessary, it also altered the receiver's sampling parameters and attempted to output received data at the new rate. Any data still in transit that had been sampled using the old parameters, was therefore output using the new sampling parameters. Audio data output using the incorrect parameters causes noise to be produced by the audio device. It was therefore necessary to transmit control information (sampling rate, number of bits, number of channels) from the sender to the receiver.

Initially a separate data packet was transmitted containing the control information. This, however, proved to be a rather simplistic solution. The reception of a control packet out of the original packet

order had disastrous results on the playback of the data stream. For example, a data packet was recorded at a rate of 22050 kHz, 8 bit and stereo. A control packet that was sent earlier but had just been received may have been for a rate of 44100 kHz, 8-bit and mono. This caused the data sample to be played at the incorrect rate, and noise was heard. A different solution needed to be found.

To ensure that the correct sampling rates are used, the control information must be sent with every packet. A data structure containing both the sampling parameters and the audio data would therefore be ideal. This would enable every data packet to be sampled using different parameters, and the WIT Audio Application would then simply alter the new parameters and output the data. No other control information would be required. This data structure is shown in figure 4.3.

```
typedef struct {
    long       rate;
    short      samplesize;
    short      channels;
    AudioDatumAudioData[MAXAUDIOSIZE];
  } DataStruct;
```

Figure 4.3. - Data structure for control information and audio data

c. Delivered QoS Synchronisation

Synchronisation of control information between client and server is necessary not only for the audio stream to be output using the correct sampling parameters, but also for the renegotiation calculations to be performed correctly.

As was discussed in the previous section, packets containing the sampling control information are delivered to the client inside a normal audio data packet. In order for accurate calculations to be performed, it is necessary to compare the measured throughput against the calculated throughput of the delivered QoS. Synchronisation between the calculation and the control information is therefore required.

After the client has performed a renegotiation of the QoS parameters, a control packet is delivered to the server containing this new information. The audio server is unaware that a change in QoS is required and continues to deliver data at the previous QoS rate. The client updates the necessary parameters, and statistics are calculated according to the new rate. Until the server receives the new QoS information, data is still received at the old rate. Increases and decreases of the QoS are

determined according to the measured network throughput. The statistics are compared against the incorrect rate and are therefore not accurate. Reliable statistics can only be measured if there is synchronisation between the data sampling parameters and the statistical functions.

New sampling parameters are only available to the application after the data packet has been transferred from the RTP daemon to the audio application. The control information is now situated on the application side of the RTP SAP message queues, and all the statistical functions requiring the information are on the RTP daemon side. The control information must be passed from the application to the daemon and cannot be stripped from the data packet before being passed across the RTP SAP by the RTP daemon.

After receiving the packet sampling parameters, the QoS Mapper is able to determine the actual QoS of the received packet. The measured throughput is compared against the bandwidth required to deliver the data packet. Any QoS renegotiations are then calculated according to this value.

If this synchronisation does not occur, the measured throughput is compared against the requested QoS. Due to the interval between the server receiving the requested QoS and the client requesting a new rate, numerous data packets may be received at the old rate and compared to the new rate. The QoS then increases or decreases every time a packet is received. This leads to an extremely high or low QoS being delivered, which will result in continuous wild fluctuations in the QoS.

After implementing the synchronisation, renegotiation only occurs when the requested QoS is equal to the delivered QoS. This takes place when the client receives the first packet sampled at the new rate. Measured throughput is compared to the requested QoS. All packets delivered at the same rate only allow a single renegotiation to occur and therefore wild fluctuations in the QoS are prevented, resulting in a smoothing effect of the delivered QoS.

4.4. Multi c as ting

As has been discussed in previous sections, the RTP Library does not currently support multicasting. A very elementary effort at adding multicasting support to the library was attempted.

Due to the fact that multicast data is transmitted to a multicast group as opposed to a particular IP address, only the RTP data receiver code needed to be changed. The only alteration needed in the sender code was to check whether the specified multicast group was a legitimate address.

The IP data receiver requires the IP address setting to be altered to INADDR ANY in order to accept any incoming messages and not only those from a specific IP address. The receiver must also check that it is receiving and transmitting data within the same multicast group. The kernel also needs to be enabled in order to join a multicast group. This requires using `setsockopt()` and specifying the following parameters:

IP NITJLTICAST_IF: the interface to be used by the multicast group,
IP MULTICAST TTL: the time-to-live, and
IP MULTICAST LOOP: the loopback interface to be used.

It is also necessary to add the IP group membership. Figure 4.4 shows the structure used for this purpose:

```
struct ipmreg
  {
    struct in_addr imr multiaddr;
      /* IP multicast address of group */
    struct in_addr imr interface;
      /* local IP address of interface */
  1;
```

Figure 4.4. — Data structure required for the addition of multicasting functionality

The above code needs to be added to the receiver whilst maintaining an unaltered sender code. The receiver and transmitter code are, however, fully inter-linked within the RTP daemon code. A single RTP socket that is responsible for both functions currently exists. A problem arose because the RTP socket cannot be easily separated between a receiver and a sender. Therefore, the multicast code needed to be implemented as part of the RTP socket, and this affected the sender as well.

A multicast class that is only implemented on the receiver is therefore required, which will cause the RTP daemon to execute normally. When the RTP session is created, the data receiver implements the necessary multicasting code, and the sender remains unaffected. This has not been introduced, as it is not directly required by the RTP Audio Application.

4.5. Conclusion

Certain key issues arose during implementation of the RTP Audio Application. These issues were directly related to the primary aim of the system: that of delivering a dynamically adjustable QoS whilst maintaining a constant information content. Resource Layer issues regarding RTP and RTCP are directly related to the delivery of the multimedia data stream.

Synchronisation and buffering of data, both at the Resource and Application layers, are crucial aspects of the system. The delivery of multimedia data, regardless of whether the QoS of the data stream is guaranteed or not, requires a close synchronisation between all layers of the system as well as the networking component. The primary synchronisation issues are related to the transfer of data between the application and media device, and across the network between the client and server. Maintaining the real-time nature of the delivered data, and therefore the delivered QoS, necessitates ensuring this synchronisation occurs at all layers of the system.

Buffering of the data is also an attempt to maintain the multimedia characteristics of the delivered data stream from the server to the client. Similarly to the synchronisation of the data stream, buffering is required between the Application Layer and media device and also between the network and Resource Layer.

Coupled with the use of RTP as the transport protocol, synchronisation and buffering ensure that the multimedia data is able to maintain the real-time characteristics of the recorded data stream. The QoS of the delivered data is therefore maintained throughout the Application and Resource layers. Although a guaranteed end-to-end QoS is not possible due to the lack of resource reservation across the network component, the maximum possible use of the available resources is ensured.

Chapter 5

QoS Renegotiation Methods

5.1. Introduction

The design of the RTP Audio Application described in this thesis provides for the real-time dynamic adjustment of the delivered QoS. A discussion of the renegotiation methods used to adjust the delivered QoS, and therefore provide an adaptive flow mechanism, is presented in this chapter.

As already discussed in Chapter 3, the Resource Monitor component of the Session Manager is responsible for implementing the specified renegotiation policy. Information, such as the received and sent timestamps of the data packet, data packet size and sequence number, is used to determine the measured throughput or loss. Depending on whether the current renegotiation method is based on packet loss or throughput, the measured value is compared against a base value. If the measured value is within an acceptable percentage of the base value, the delivered QoS is increased. If it is outside the maximum acceptable value, the QoS is reduced.

An in-depth investigation of the measured statistics, the different renegotiation methods and the general renegotiation algorithm, is presented in this chapter.

5.2. QoS Groups

Within the QoS renegotiation algorithms for the RTP Audio Application, a QoS group is defined as a collection of all elements in the QoS table that require the same bandwidth to deliver a unit time of data. The grouping of QoS elements in this manner is necessary in order to provide a differentiation mechanism allowing for the prioritisation of elements within the data table.

Figure 5.1 shows an example of a QoS group for the bandwidth value of 44100 Bps:

Sample Rate	Sample Size	Channels	Bandwidth
11025			44100
22051	1		44100
22050			44100
44100	1		44100

Figure 5.1. - QoS group for a bandwidth value of 44100 Bps

As can be seen from Figure 5.1, four different QoS elements require the same bandwidth to deliver a unit time of data. Therefore, from a Resource Layer aspect there is no reason to choose one value over another. The user is, however, able to differentiate between items. Different sampling parameters exhibit certain characteristics, and depending on the nature of the application, it may be necessary to choose one value over another. The prioritisation of items within the table is discussed in detail in section 5.2.

Not only are QoS groups important at a Resource and User level, but also for performing a renegotiation of the QoS. A renegotiation of the QoS causes the delivered QoS to increase or decrease by a stipulated number of QoS groups. The required bandwidth for a QoS group is double the bandwidth required for one group lower. Conversion between a QoS group and a QoS element in the QoS table is performed using the following calculation:

$$\text{QoS Group} = \log_2 \left| \frac{\text{(bandwidth value of QoS Element)}}{\text{(lowest bandwidth value)}} \right|$$

5.3. Statistics

Quantitative assessment of the various renegotiation methods is only possible if accurate, reliable data is available. A discussion of the elements that are required for processing the data, as well as the statistical formulae is presented in the following section. The results of these calculations and the analysis of the results is presented in Chapter 6.

5.3.1. Statistical Elements

Calculations can only be performed using elements that are delivered in the RTP packet from the sender to receiver, or elements that are calculated after receiving the data packet. The RTP daemon receives the entire packet and performs the necessary network maintenance. The RTP packet is separated from the RTP header and then stored in a memory buffer.

After the IP, UDP and RTP headers are stripped from the data packet, the packet size, timestamp and sequence number are passed to the Resource Monitor, where a calculation of packet loss is performed. Late packets are determined by comparing the expected inter-packet arrival time to the actual inter-packet arrival time. Lost and out-of-order packets, however, require a comparison of the current and previous packet sequence numbers. The data table is used to maintain a history of all received data packets. The sequence number, received and sent timestamps, and the data size (in bytes) are stored in the data table.

The following four elements are used for calculating the relevant statistics:

1. Timestamp

   Two timestamps are required to calculate the necessary statistics. The delivered timestamp is calculated by the data sender and sent in the RTP header. The Resource Monitor at the receiver calculates the received times tamp.

2. Sequence Number

   A random sequence number is generated for the first packet that is delivered and is increased sequentially for each subsequent packet. The sequence number is also delivered in the RTP header.

3. Packet Size

   The packet size is calculated after the headers have been stripped from the data packet. A calculation is performed to determine the amount of memory required for the data to be stored in a memory buffer. This calculation includes the control information transmitted with each data packet. The number of bytes required for the control information is subtracted from this value.

4. Control Information

QoS parameters are delivered with each data packet. Although these parameters are not stored for every packet that is received, any renegotiation that occurs requires the new sampling parameters to be recorded.

5.3.2. Statistical Calculations

Although the QoS renegotiation is determined by the measured throughput or packet loss, it is necessary to measure a number of extra statistics in order to accurately assess the success of the different renegotiation methods. The success of the particular renegotiation method is determined by the measured throughput, packet loss and jitter values. The discussion of the results in Chapter 6, concentrates primarily on these calculations.

In order to present a more thorough investigation, extra formulae are also discussed in the following section. The resulting values may also be required to distinguish between methods that exhibit similar throughput and jitter characteristics.

The three primary statistics (throughput, loss and jitter) are investigated first, followed by a discussion of packet arrival rate, QoS ratios and packet size analysis.

5.3.2.1. Throughput

Total Throughput

$$\text{Throughput} = \frac{\text{Total bytes received}}{\text{Time Period}}$$

The number of bytes successfully transferred in unit time over a connection on a sustained basis is known as the total throughput. Depending on the renegotiation method, the unit time can be either the time required to transfer a single packet *(Current Throughput),* the total length of the connection *(Total Throughput)* or the interval over which the renegotiation is measured *(Smoothed,* or *Congested Throughput).*

Average, Minimum and Maximum Throughput

Renegotiation of the QoS is determined by the current throughput value. An indication of the overall throughput for the entire lifetime of the connection is determined by the average throughput value. The minimum and maximum throughput is an indication of the minimum and maximum number of bytes delivered across the connection over the defined unit time.

5.3.2.2. Packet Loss

Total Packet Loss

$$\text{Total Packet Loss} = \frac{\text{Total packets lost}}{\text{Total packets delivered}}$$

The Total Packet Loss is the percentage of the number of packets received at the receiver to the number of packets delivered by the server.

Figure 5.2 describes the three situations where a packet can be defined as lost:

Situation
Packets that are sent by the server but never arrive at the receiver
Packets that arrive out of order and the preceding packet has already been played
Packets that arrive in sequence but are "too late" to be played. This is dependent on the defined jitter formula.

Figure 5.2. - Three situations leading to packet loss in the RTP Audio Application

Renegotiation methods based on packet loss include loss measured in all three situations. The actual throughput measurement is calculated as bytes received per second, and the packet loss measures the number of packets successfully received versus the number of packets transmitted. It is important to provide a breakdown of the total packet loss, as different renegotiation methods may continuously suffer from a particular type of loss.

Lost Packet Ratio

$$\text{Lost Packet Ratio} = \frac{\text{Total packets not arriving at receiver}}{\text{Total packets delivered}}$$

This ratio reflects situation (1) in figure 5.2 above. The sequence number for the previous packet is subtracted from the sequence number for the current packet. If the difference is greater than one, then it indicates the number of packets that have been lost.

Out of Order Packet Ratio

$$\text{Out of Order Packet Ratio} = \frac{\text{Total out of order packets}}{\text{Total packets delivered}}$$

This ratio reflects situation (2) in figure 5.2 above. The sequence number for the previous packet is subtracted from the sequence number for the current packet. If the difference is less than zero, the current packet has arrived out of order and must be discarded. Even though the packet has arrived at the receiver it must be discarded from the data table. Only data that is output at the multimedia device is included in any statistical calculations.

Late Packet Ratio

$$\text{Late Packet Ratio} = \frac{\text{Total late packets}}{\text{Total packets delivered}}$$

This ratio reflects situation (3) in figure 5.2 above. Jitter is the variation in the packet arrival time of data packets [Schulzrinne *et. al.* 1996]. A packet is considered to be too late for output if received after an acceptable time. The acceptable time is defined BY the particular renegotiation method. The expected time is the interval between the current packet timestamp and the previous timestamp. A late packet is therefore a packet that has been received in the correct order, but the inter-packet arrival time is too large for the packet to be output at the multimedia device.

5.3.2.3. **Jitter**

According to Schulzrinne [1996], "Jitter is the estimate of the statistical variance of the RTP data packet inter-arrival time, measured in timestamp units and expressed as an unsigned integer. The inter-arrival jitter is defined to be the mean deviation (smoothed absolute value) of the difference, D, in packet spacing at the receiver compared to the sender for a pair of packets. This is equivalent to the difference in the "relative transmit time" for the two packets".

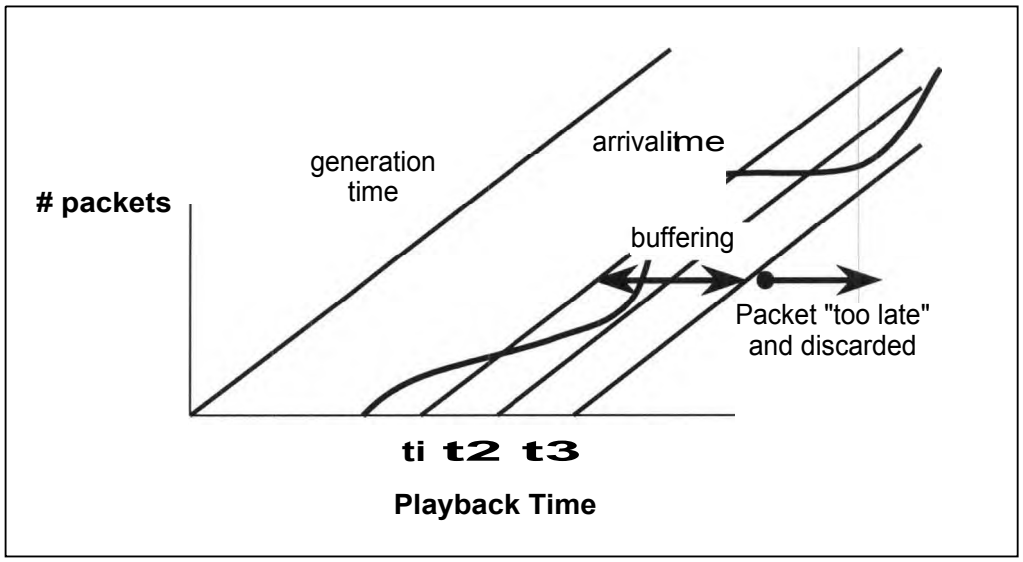

Figure 5.3. - Jitter regulation by buffering data [Campbell *et. al.* (2)]

Packets received before t3 (in figure 5.3.) are buffered at the receiver and then output in sequence. Data packets that are received after t3 are considered "too-late" to be output and are discarded.

Jitter Calculation

Expected time$_{(i,j)}$ = sent timestamp$_{(i)}$ - sent timestamp$_{(j)}$

Actual time$_{(i,j)}$ = received timestamp$_{(i)}$ - received timestamp$_{(j)}$

Difference$_{(i,j)}$ = expected time$_{(i,j)}$ - actual time$_{(i,j)}$

Jitter = Previous jitter value + ((absolute value(Difference$_{(i,j)}$)
- Previous Jitter Value) / (gain parameter))

Schulzrinne defined the above jitter formulae and recommends a gain parameter (1/16 in the above calculation) to give a good noise reduction ratio while maintaining a reasonable rate of convergence [1996].

Assuming a constant packet size, the inter-packet arrival time is directly related to the delivered QoS. Delivery of a 1024 byte packet at a QoS of 900 Bps requires an inter-packet arrival time of approximately 1200 milliseconds. The higher the delivered QoS, the lower the expected inter-packet arrival time. Jitter is directly related to the inter-packet arrival time, and therefore indirectly related to the delivered QoS. If all packets are received perfectly on time, then the measured jitter will equal zero. The larger the difference between the expected and actual inter-packet arrival times, the higher the measured jitter value.

Average Inter-Packet Arrival Time

$$\text{Average inter-packet arrival time} = \frac{E\,(\text{measured inter-packet arrival times})}{(\text{number of packets received} - 1)}$$

The inter-packet arrival time is the actual time (measured in milliseconds) between the arrival, at the receiver, of subsequent data packets. Although the packet size is not altered when the QoS is renegotiated, the sampling interval is affected. The higher the delivered the QoS, the shorter the sampling interval. The average inter-packet arrival time calculation is an indication of the rate that the packets are received by the client and does not take into account the "lateness" of the packet, i.e.: how long after the expected time the packet is actually received.

5.3.2.4. Packet Arrival Rate

$$\text{Packet Arrival Rate} = \frac{\text{Number of packets received in period}}{\text{Renegotiation period}}$$

Packet Arrival Rate is measured as the number of packets received per second. This value is calculated over the renegotiation period. No calculation is performed for the *Current* or *Total* renegotiation methods, as the measurement period is a single packet.

Average Packet Arrival Rate

$$\text{Average Packet Arrival Rate} = \frac{\text{(Packet Arrival Rates)}}{\text{Number of renegotiations}}$$

The Average Packet Arrival Rate is provided as a means of comparison between different renegotiation methods. Lost and out-of-order packets are not included in the calculation, but late packets are included. If a packet is lost, then the previous packet will cause the time (but not the number of received packets) to increase. Lost packets therefore affect the average packet arrival rate adversely.

5.3.2.5. QoS Ratios

Maximum QoS Packet Count

The maximum QoS packet count is the number of packets received before the maximum QoS is delivered. This is an indication of how quickly the renegotiation method is able to adapt to available network resources. The lower the value, the faster the renegotiation method is able to adapt.

QoS Decrease Ratio

$$\text{QoS Decrease Ratio} = \frac{\text{Number of QoS decreases}}{\text{Total renegotiations performed}}$$

The ratio of the number of times the QoS is decreased to the number of renegotiations that are performed is known as the QoS decrease ratio. If the QoS is constantly fluctuating, then the user is subjected to a constant change in the received audio or video quality.

QoS Increase Ratio

$$\text{QoS Increase Ratio} = \frac{\text{Number of QoS increases}}{\text{Total renegotiations performed}}$$

The QoS increase ratio is similar to the QoS decrease ratio, except the number of increases in the QoS is measured.

5.3.2.6. Packet Size Analysis

The RTP Audio Application does not alter the delivered packet size during the lifetime of the connection, regardless of the requested QoS. If a change in QoS occurs, then the sampling interval is adjusted accordingly.

Every packet requires a network protocol header (the RTP Audio Application uses IP), a UDP header and an RTP header (total: 40 bytes). As discussed in section 4.3.3.2, control information is also delivered with each packet. The RTP Audio Application requires a sampling rate (long — 4 bytes), sample size (short — 2 bytes) and the number of channels (short — 2 bytes). Control information requires 8 bytes, which brings the overhead on each data packet to 48 bytes.

The shorter the packet size, the larger the percentage of the throughput used to deliver the packet overhead. A larger packet size requires more resources to process and deliver the packet, which results in a trade-off between resources and throughput. Packet Size analysis attempts to find the ideal data packet size for each renegotiation method by quantitatively examining the results of all the measured statistics.

5.4. Renegotiation Methods

5.4.1. Current Throughput

The *Current Throughput* renegotiation method attempts to deliver the correct QoS based on the current network conditions. No corresponding Packet Loss method is described, as it is impossible to calculate a loss statistic for a single point in time, loss is only measured over a continuous scale.

After receiving a data packet, the client calculates the current throughput value. A renegotiation of the QoS occurs according to the measured throughput value. If the measured throughput is within an acceptable percentage of the delivered QoS, the QoS is increased by one QoS group. If the throughput is not within the specified acceptable percentage, the QoS is decreased to the QoS group that is closest to the measured throughput value.

For example, assume that a QoS of 22050 Hz, 8-bit, mono is currently being delivered. The delivered throughput required is therefore 22050 bytes per second. If this 22050 byte sample takes two seconds to be delivered, this indicates a current throughput of 11025 Bps. A QoS renegotiation based on the

measured value is then performed. The client delivers a control packet to the server requesting the QoS to be adjusted to 11025 Bps.

The *Current Throughput* method performs a QoS renegotiation after each packet is received, and this causes the delivered throughput to experience the largest fluctuation over time. Although all available network resources are continuously utilised, the wild fluctuations in the delivered QoS are a major disadvantage. Although the QoS can only increase by a single QoS group per renegotiation, the QoS can be adjusted rapidly from the minimum to the maximum value. If there are five QoS groups, then it is possible to achieve maximum QoS in five renegotiations. Conversely, the QoS can be reduced from maximum to minimum QoS in the same number of renegotiations.

5.4.2. Total Throughput and Total Loss

The *Current Throughput* method measurements are able to provide a "snapshot" of the network conditions at a particular instance in time. More relevant to this thesis is the state of the network over a period of time. The Tota/ renegotiation method attempts to measure the throughput or packet loss over the entire lifetime of the connection. Naturally, whilst a connection is "active", the renegotiation period will extend from the time of the first packet being received until the current time.

Throughput is measured as the total bytes received divided by the length of the connection. The throughput value is calculated after each packet, and this value is then used as a lookup into the QoS table. If the measured throughput is within the defined acceptable percentage of the delivered throughput, then the delivered QoS is increased by a single QoS group. If a decrease in QoS is required, the QoS closest to the measured total throughput value is requested. Although wild fluctuations in delivered throughput are experienced using the *Current Throughput* method, the received throughput is stable using the Total method. The delivered QoS fluctuates constantly, however.

Consider the following case: an initial QoS of 44100 Bps is delivered across a lightly loaded network. As packets are delivered on time, the throughput gradually increases, until an improvement in the QoS is required. The delivered QoS is increased to 88200 Bps. The measured throughput is, however, close to 44100 Bps. After a single data packet of data sampled at 88200 Bps is delivered, the total throughput is recalculated. Although the 88200 Bps packet will increase the throughput slightly, it will not increase to within an acceptable percentage of the new QoS. The QoS is then reduced to the value that is closest to the measured throughput value (i.e. 44100 Bps). After receiving this packet, a recalculation of the throughput is made and it is found to be within the acceptable percentage of the QoS. An increase to 88200 Bps is necessary. This fluctuation in delivered QoS is as unacceptable as the *Current Throughput*

method. Continuous increases in QoS as described above are extremely rare, and the higher the initial QoS, the higher the average throughput for the entire connection.

All renegotiation methods based on loss differ slightly from the throughput methods. Due to the fact that there is no direct mapping between packet loss and delivered QoS, two loss thresholds, a minimum and maximum, are defined. If the measured loss is less than the minimum acceptable loss threshold, then the packet loss is considered sufficiently low for an increase in QoS to occur. Similarly, if the loss is greater than the maximum loss, a reduction in QoS is requested. The major difference between the packet loss and throughput methods is that if the measured loss is greater than the minimum loss, but less than the maximum loss, no renegotiation occurs. The lack of a direct mapping between loss and delivered QoS prevents a meaningful reduction in QoS from occurring. By using loss thresholds, it is possible for the user to define the maximum and minimum percentage of lost packets.

Using the *Total Loss* method, fluctuations in QoS are either extremely slow or extremely rapid. If the measured loss is greater than the maximum acceptable loss percentage, the delivered QoS is continually reduced until the maximum threshold is reached. At this point, the current delivered QoS is maintained until the packet loss percentage is less than the minimum loss threshold. Depending on the set threshold levels, this may occur over a long period. At the time, when the minimum threshold is reached, the QoS is continually increased until a high packet loss percentage is reached. The QoS is therefore increased to a value that is too large for the current network conditions. Once this value is reached, large packet loss occurs and the total measured loss quickly increases. The QoS rapidly decreases until the minimum QoS is delivered, and the process is repeated.

5.4.3. Smoothed Throughput and Smoothed Loss

Another solution, based on both a current and a total measurement, is clearly required. Although the current network-state is important, the delivered QoS must not be altered so frequently that it is no longer applicable. Similarly, the *Total* method provides a history over the lifetime of the connection, and trends can therefore be noticed.

The *Smoothed* QoS renegotiation method introduces new two concepts. Firstly, a **smoothing value** determines the frequency with which the connection will be monitored. The smoothing value can either be measured as a time period, the number of packets, or the number of bytes received. Secondly, a **smoothing fluctuation** variable determines the amount of fluctuation that is allowed at each renegotiation. A smoothing fluctuation of two will allow the renegotiation to increase or decreases the QoS a maximum of two QoS groups.

Current network state information that prevents wild fluctuations in QoS is clearly an advantage for the *Smoothed* renegotiation method. If the smoothing fluctuation is set to a high level, such as three or four, then large fluctuations may still occur. The smoothing variables are user-definable, which allows the user to exert some control over the renegotiation process.

Both of the previous two methods *(Current* and *Total)* can be simulated using the *Smoothed* method. A smoothing fluctuation of four (maximum) and a smoothing time period of close to zero will have the same effect as the *Current* method. A maximum time period and fluctuation level will cause the system to act in a similar manner to the *Total* method. This illustrates a major problem with the *Smoothed* method; if the variables are set to incorrect values then any noticeable advantages may be lost.

5.4.4. Congested Throughput and Congested Loss

The *Congested* method is based on the work by Busse *et. al.* [1995],and is similar to the *Smoothed* method described in the previous section. The major difference between the *Smoothed* and *Congested* methods is that the *Congested* method defines three state-variables (congested, loaded and unloaded) whereas the *Smoothed* method does not define a state-variable. An unloaded state indicates that network usage is very low, and a high QoS can be delivered. Loaded indicates a more heavily used network, and fluctuations in the QoS may be necessary to maintain this state. If a congested state is achieved, then the network is largely unusable.

The state-variable determines the amount of fluctuation allowed in the delivered QoS. An unloaded state is optimal and the system will always attempt to deliver a QoS that will allow the unloaded state to be achieved. Whilst the system is in an unloaded state, only minor fluctuations in the QoS are permitted, which should prevent temporary network conditions from affecting the QoS too greatly. A fluctuation of one will allow the throughput to be either halved or doubled and this should be sufficient to maintain the unloaded state.

A loaded state indicates a more permanent deterioration in network conditions. Due to the increased load, transient increases and decreases in available resources are likely to occur and a more flexible policy is therefore required. A smoothing fluctuation value of two is more appropriate while in this state. Changes in the available network resources are reflected by the fluctuation in the delivered QoS. As already discussed, the ideal state for the system to be in is an unloaded one. Allowing for a larger fluctuation whilst in the loaded state ensures that *if* network conditions improve, then a return to the unloaded state can be rapidly achieved.

The congested threshold is set at such an undesirable QoS that the user would rather lose packets than receive data at the specified QoS. The congested state is never entered, and this defines the minimum QoS that is to be delivered. If the measured QoS is less than the congested threshold, the QoS is reduced to the closest value to the threshold.

The smoothing time-period ensures that if the network resources suddenly deteriorate, an appropriate adjustment in the QoS will occur rapidly. No renegotiation in the QoS may occur until the smoothing time-period has completely elapsed. Fluctuations in network resources during this time-period are ignored and the values are measured for the entire period.

The major problem that may be encountered using the *Congested* method is if the state values are set as either too high or too low. The unloaded state may never be achieved or the smoothing time-period may be so long that it is not indicative of the current conditions.

5.4.5. Adjusted Method

After performing numerous experiments and observing the advantages and disadvantages of each method, the *Adjusted* method was created. In experiments conducted using the previous methods, rapid increases in QoS led to high packet loss occurring. To prevent this high packet loss, the *Adjusted* method is only able to increase the QoS by a single QoS group. Decreases in QoS are, however, dependent on the current network-state.

The network state values, and the corresponding allowed fluctuation were also changed. The more unloaded the state, the higher the fluctuation allowed. The *Congested* method attempts to deliver the highest possible QoS continuously, whereas the *Adjusted* method attempts to reduce the measured packet loss. The change in the smoothing fluctuation value is an attempt to rapidly reduce the delivered QoS in order to minimise packet loss. The state is determined by the measured throughput, however, not the packet loss.

State	Throughput	Fluctuation
Unloaded	>= 7400 Bps	4
Loaded	1000 > x < 7400	2
Congested	<= 1000 Bps	4

Figure 5.4. - Network-state and allowed fluctuation for the Adjusted method

The fluctuation values shown in figure 5.4, and the throughput and loss values in figure 5.5, are based on estimates of the optimum values for the RTP Audio Application observed during the experimentation process. These values are user adjustable, and depending on the type of multimedia data delivered, may require adjusting.

With the *Adjusted* method, increases and decreases in QoS are no longer dependent on only the throughput or loss. An increase in QoS is possible only if the measured throughput and packet loss during a renegotiation period are both within the specified acceptable window. A reduction in the QoS occurs if either the loss or throughput are outside the maximum acceptable window. Similarly to the renegotiation methods based on packet loss, the QoS can be maintained if it is not increased or decreased. The following values were used for the experiments:

Renegotiation Performed	Throughput	Packet Loss
Increase • °S...	<= 10%	0.1
Maintain QoSi j	10`)/0 < x < 25%	0.1 < x < 0.2
IIMMMW: j	>= 25%	
1		

Figure 5.5. - Throughput and Packet loss percentages and renegotiation performed

5.4.6. Description of QoS Renegotiation Algorithm

In the following section, the basic structure of the algorithm that is used for all renegotiation methods is discussed. Code used in both the throughput and loss method is identified with "Throughput and Loss :", otherwise the code is identified as either a "Throughput" or "Loss" method. Psuedo-code based on C++ is presented in this section. Appendix 2 contains the full code listing for all of the renegotiation methods.

The renegotiation method algorithm is initially presented in full (figure 5.6.), with no regard to specific differences between the loss and throughput methods. Subsequent to this primary overview of the algorithm, a step-by-step analysis of the algorithm is discussed and particular differences between the loss and throughput methods are considered.

Step 1-Check for elements in the data table.

Step 2 - Define the acceptable variables for an increase or decrease in QoS.

Step 3 - Define the maximum fluctuation.

Step 4 - Check the delivered QoS.

Step 5 - Calculate the elapsed time.

Step 6 - Calculate the measured value.

Step 7 - Add the measured value to the statistics table.

Step 8 - Increase or Decrease the QoS.

Step 9 - Adjust delivered QoS.

Step 10 - Prioritise the delivered QoS.

Figure 5.6. — General QoS renegotiation algorithm

Step 1- Check for elements in the data table:

```
Throughput and Loss:
        if (no elements in the data table)
          return 0;
```

It is necessary to check if there is an item at the first position in the data table (discussed in section 5.3.1.). If there is no item present, then it is not possible to perform any renegotiation.

Step 2 - Define the acceptable variables for an increase or decrease in QoS:

```
Throughput:
        Acceptable percentage = 0.10;
Loss:
        Acceptable loss = 0.05;
        Minimum loss = 0.25;
```

As previously discussed for the *Adjusted* method in section 5.2.6, the threshold values used throughout the experimentation section are ideal for the specific RTP Audio Application under observation. Other data types, such as a video stream, may require different threshold values.

An acceptable throughput/loss variable determines whether any renegotiation should occur. Throughput methods require a single variable. If the measured throughput is within an acceptable percentage of the delivered throughput, then the QoS is increased by one QoS group. Failure to deliver a throughput within the acceptable amount results in a reduction of the delivered QoS to that which is closest to the measured throughput value.

A measured loss cannot be mapped directly onto a QoS value because loss is dependent on both the current network conditions and the delivered QoS. An extra variable is required for a loss calculation. If the measured total loss is greater than the minimum allowed loss, the QoS is reduced by a single QoS group. If the measured loss is between the minimum and maximum loss values, no renegotiation occurs and the delivered QoS value is maintained.

Step 3 - Define the maximum fluctuation:

```
Throughput and Loss:
      Max fluctuation = 1;
```

Each renegotiation method defines a maximum fluctuation value. Increases or decreases in the negotiated QoS cannot be larger than max `fluctuation` QoS groups.

Step 4 - Check the delivered QoS:

```
Throughput and Loss:
      if (QoS of received packet = QoS after renegotiation)
```

After the renegotiation of the QoS has occurred, a control packet is delivered to the data server requesting a specific QoS. The server then begins delivering data packets at the new QoS. If the delivered QoS parameters are equal to the measured QoS parameters, this is the first packet the server has delivered after renegotiating the QoS. If this is not true, then it may be possible that a renegotiation has already occurred and the control packet is currently in transit to the server. Multiple renegotiations based on the same measurements could then occur.

Depending on the type of renegotiation, throughput or loss, specific values are reset to zero at this point. For example, the *Congested Throughput* method requires a new throughput calculation for each

renegotiation period. The time and throughput values will therefore be reset at this point and a new calculation performed.

Step 5 - Calculate the elapsed time:

```
Throughput:
      time   received timestamp of current data packet -
             received timestamp of previous data packet
```

If the time elapsed is equal to zero, the throughput cannot be calculated, which means that the increase or decrease in QoS cannot be determined.

Step 6 - Calculate the measured value:

```
Throughput:
      throughput = bytes received in renegotiation period /
                   time of renegotiation period
Loss:
      loss = total packets lost / total packets received
```

Total packet loss and total packets received are variables within the Resource Monitor and are calculated whilst checking for any packet loss. These variables are then used to calculate the measured throughput or packet loss for the renegotiation period.

Step 7 - Add the measured value to the statistics table:

```
Throughput and Loss:
      Add throughput value to statistics table
      Add loss value to statistics table
```

The Statistics Manager stores both the value and the timestamp at which the value was measured. When the application is closed, all values are written to data files, which are then used to graph the measured values.

Step 8a - Increase the QoS:

```
Throughput:
      if (throughput is within an acceptable percentage of
          the expected throughput)
        { increase the QoS }
Loss:
      if (loss < minimum allowed loss)
        { increase the QoS }
```

For each renegotiation method a max `fluctuation` variable defines the maximum number of QoS groups that the QoS can be increased or decreased by.

The limit determining whether the QoS will increase or decrease is calculated. The QoS increases if the measured throughput is within an acceptable percentage of the actual throughput. If the loss is less than the acceptable loss, an increase of the delivered QoS of the defined number of QoS group will occur. QoS groups are discussed in section 5.2.

Step 8b - Decrease the QoS:

```
Throughput:
      if (throughput is outside an acceptable percentage of the
          expected throughput)
        { decrease the QoS }
Loss:
      if (loss > maximum allowed loss)
        { decrease the QoS
```

If the throughput does not increase, then a decrease in QoS will occur. The new QoS is equal to the lower of the previously measured throughput, or previous QoS group, minus the maximum fluctuation allowed.

Loss methods do not necessarily change the delivered QoS. If the measured loss is between the acceptable and minimum values, no renegotiation occurs. A reduction in the QoS occurs if the measured packet loss is greater than the minimum allowed loss.

Step 9 — Adjust delivered QoS

```
Current and Total hroughput and Total Loss:
      Find the QoS that is the nearest, but lower than, the
      measured thr ughput
Other methods:
      Reduce the QoS by the maximum fluctuation value
```

The *Current* and *Total Throughput* methods do not use QoS groups to perform a decrease in the QoS. Instead, the closest bandwidth val ie lower than the measured throughput is used as the new QoS value. The *Current* method attempts to imulate the prevailing conditions and adapt to changing conditions immediately.

If the measured throughput is not within the acceptable percentage of the actual throughput, the QoS is reduced to that which is closest to the actual value of the measured throughput. The QoS can never be reduced to zero. The minimum that it can be reduced to is QoS group 0, in other words, the lowest possible entry in the QoS table, which in the RTP Audio Application is 11025 Bps.

Step 10 — Prioritise the delivered QoS:

```
Throughput and Loss:
      Find the element in the QoS table with the highest
      priority.
      This is the new delivered QoS value.
```

After calculating the correct QoS group, it is necessary to find the most desired value within the group. Prioritisation of the QoS elements within the QoS table allows this to be easily achieved.

5.5. Closing Remarks on QoS Renegotiation Methods

The renegotiation methods discussed in this chapter are the final component of the experimental RTP Audio Application. The layered design discussed in chapter 3 provides a component-based framework that allows additional sections to be added to the system without requiring a complete redesign. All renegotiation methods are implemented as part of the Resource Monitor component of the Session Manager.

Chapter 4 discussed all aspects of the implementation relevant to this thesis, other than those relating to the dynamic adjustment of the delivered QoS. This vital component of the experimental system was discussed in detail throughout this chapter. A general renegotiation algorithm was deliberated upon, from which all renegotiation methods were developed. All of the different methods were then discussed individually, including any changes required from the base algorithm.

An experimental testbed is discussed in the following chapter, including the measured results and statistics, in an attempt to discover the advantages and disadvantages of each renegotiation method. All statistical formulae reviewed earlier in this chapter are measured. A comparative series of results are provided allowing conclusions to be drawn as to the suitability of each renegotiation method to the particular traffic type that is currently simulated. A quantitative assessment of each renegotiation method and it's corresponding ability to provide for a dynamic adjustment of the delivered QoS whilst maintaining a constant information content is therefore the primary aim of Chapter 6.

Chapter 6

Experiments and R e s ults

6.1. Introduction

After initial test experiments were conducted using the Rhodes University local-area network, it became apparent that a new experimental testbed would have to be used. Across this local-area network, a maximum throughput of 10 Mbps (1.25 MBps) is possible. The maximum rate that is required to deliver CD quality audio is 176400 Bps. Regardless of the amount of traffic present on the network segment, it would be difficult (if not impossible) to test the renegotiation methods correctly due to this size constraint. It was therefore necessary to find an IP network that would be able to limit the delivered QoS, yet also provide enough available bandwidth to correctly assess the different renegotiation methods.

A serial line running a PPP daemon was used for the testbed IP network. The network connection using the PPP connection was free of other traffic and full control of the resource therefore possible. A major disadvantage, however, is the maximum speed of the line. Although the line speed is configurable, the highest possible speed is only 115200 bps. This equates to approximately 14 kBps, which is slightly higher than the bandwidth required for delivery of audio data at a rate of 11025 Hz, 8-bit, mono.

OSS permits audio sampling rates below 11025 Hz. It was therefore necessary to alter the minimum and maximum sampling rates for experiments using the PPP line. In order to maintain the five QoS groups that were used previously, it was decided to set a minimum sampling rate of 900 Hz and a maximum of 3600 Hz. The highest bandwidth value would then correspond with the maximum theoretical delivery rate. The new mapping table for the RTP Audio Application (figure 6.1.) appeared as follows:

Sample Rate	Sample Size	Channels	Bytes/sec
900	1	1	900
900	1	2	1800
900	2	1	1800
900	2	2	3600
1800	1	1	1800
1800	1	2	3600
1800	2	1	3600
1800	2	2	7200
3600	1	1	3600
3600	1	2	7200
3600	2	1	7200
3600	2	2	14400

Figure 6.1. - Mapping table illustrating the bandwidth values for the RTP Audio Application (adjusted for the PPP line)

6.2. Experiments

In order to test the different renegotiation methods described in chapter 5; a series of five experiments was devised. Each experiment was intended to measure a different type of network traffic. This would enable an overall picture of the strengths and weaknesses of the different methods to be formulated so those particular attributes of each method could then be identified.

Simulation of the network traffic was achieved by using several simultaneous "ping" programs. Ping is a simple network tool that delivers a specified amount of data (known as the payload) to a specified host. Upon receiving this data packet, the receiver returns a confirmation packet to the sender. When this confirmation packet is returned, the sender of the original packet is informed that the receiver is "alive".

Amongst other things, ping allows the size of the payload to be defined, the quantity of packets to be delivered and the interval between packets. Using several simultaneous ping programs with a reasonable payload size (the default is 56 bytes), it is possible to saturate a network line with a small maximum bandwidth. By varying the number of simultaneous ping programs, it is possible to simulate different traffic patterns.

Although the maximum theoretical transfer rate across a PPP serial line is 14400 Bps, measurements have shown that this is not the case. During the experimentation process, the maximum achieved

throughput was measured at approximately 10000 Bps. Complete saturation of the PPP line is achieved by executing 35 simultaneous ping programs each with a payload of 256 bytes (and 28 bytes for the IP and ICMP headers) 35 * (256+28) = 11360 Bps. This does not take into consideration any traffic from the RTP Audio Application. A maximum attempted saturation of 20 simultaneous pings (20 * (256+28) = 5680 Bps) provides sufficient remaining resources for the delivery of a small quantity of audio data at the same time.

6.2.1. No-Traffic Experiment

The first experiment was conducted without any other network traffic on the line in order to provide baseline values for all renegotiation methods. The baseline values are then compared to the measured data values from the other experiments.

6.2.2. Experiment 1

The first experiment checks the renegotiation method's ability to deal with a linear increase in network traffic (up until the point that the line reaches near-saturation) followed by a linear decrease to zero traffic. At time zero, the experiment was started. Two simultaneous pings were started (sending 256 byte payloads at 1-second intervals). After every 10 seconds, two more were added until saturation point was reached (20 simultaneous). A linear decrease then occurred, with two pings dying every ten seconds until zero remained.

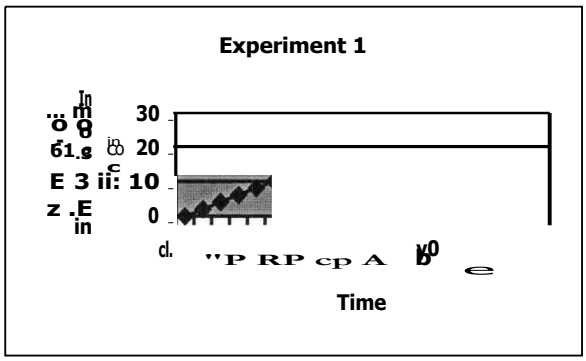

Figure 6.2. - Graph showing network saturation over time (Experiment 1).

6.2.3. Experiment 2

Experiment 2 attempted to determine how well each method was able to react to a sudden increase in network activity. For 50 seconds, saturation of the network occurred (20 simultaneous pings delivering 256 bytes at 1-second intervals). Zero traffic for the following 30 seconds allowed the network to return to a normal state, followed by another 50 seconds of saturation.

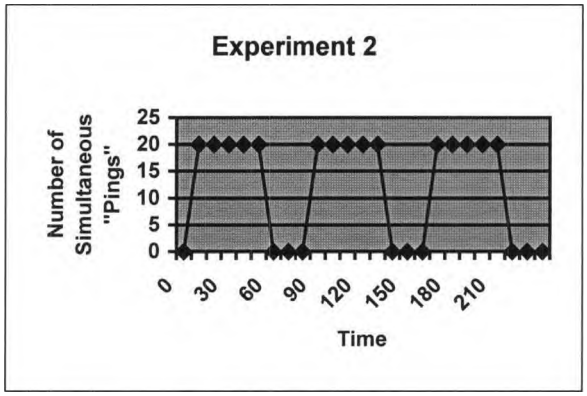

Figure 6.3. - Graph showing network saturation over time (Experiment 2).

6.2.4. Experiment 3

Delivery of a constant, high bandwidth was simulated during Experiment 3. Saturation of the line was not attempted, but a level equal to approximately half-saturation was maintained for the entire duration of the connection. Ten simultaneous pings (10 * 256 = 2560 Bps) ensured that a high level of network traffic existed on the network.

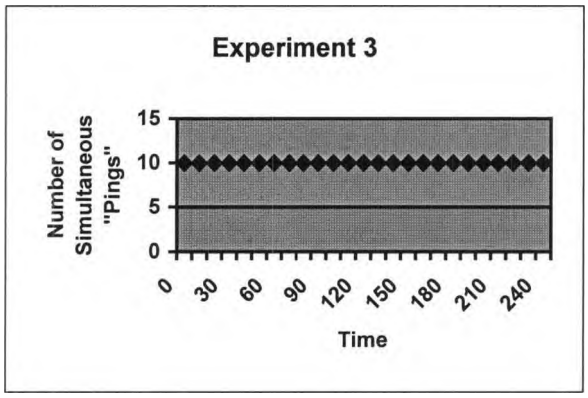

Figure 6.4. - Graph showing network saturation over time (Experiment 3).

6.2.5. Experiment 4

An attempt at simulating random network traffic was the aim of Experiment 4. Although it is not possible to accurately recreate random traffic, simulation of varying amounts of available network resources is possible. Every 10 seconds the number of simultaneous pings was altered and this caused the available network bandwidth to change accordingly. The experiment aims to measure the ability of each method to adjust to fluctuations in available resources.

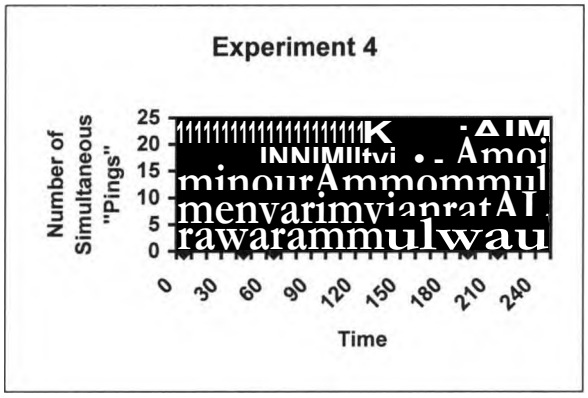

Figure 6.5. - Graph showing network saturation over time (Experiment 4).

6.3. Throughput Renegotiation Methods

The results of the experiments from the four different throughput renegotiation methods, the *Total, Current, Smoothed* and *Congested* methods, are discussed in the following section. These results are presented according to the statistical formulae groups discussed in Chapter 5, namely throughput, jitter packet loss and QoS. An analysis and discussion of the measured statistics is presented and a comparison of all the throughput methods investigated.

6.3.1. Total Throughput Method

The *Total Throughput* method delivered a maximum initial QoS (14400 Bps), whereas all other methods delivered a minimum initial QoS (900 Bps). If a minimum initial QoS is delivered, the *Total Throughput* method is unable to increase the QoS, and this minimum is then delivered for the entire connection-time.

Throughput

l'ixperimen t Number	Average Throughput	Minimum Throughput	Maximum Throughput
No-Traffic'-' ..	7435.414	7325.41	i0375.5
Ex.eriment 1	5704.972	4466.6	i0400.0
Ex.eriment 2	3648.357	3323.21	9475.05
Ex.eriment 3	5884.361	5075.13	i0375.3
Ex eriment 4	5273.563	4316.95	i0400

A high average throughput (7435 Bps) is achieved across an unloaded line. This is comparable to the measured throughput for other throughput methods. The delivered QoS constantly fluctuates between 7200 Bps and 14400 Bps, which ensures a high throughput. If the measured throughput is greater than 6480 Bps (the acceptable throughput window), then the QoS is increased to 14400 Bps. Due to the slow increase in throughput, the new QoS cannot be maintained and is then reduced to the highest QoS that is lower than the measured throughput (7200 Bps). This repetitive increase/decrease occurs throughout the entire connection.

Rapid decreases in the delivered QoS are possible during the initial connection period. As time progresses, changes in throughput (and delivered QoS) occur far slower. The delivered QoS is maintained regardless of network conditions during most of the experiments. Only during periods of complete network saturation is the throughput reduced sufficiently to lower the delivered QoS. Due to this slow change in delivered QoS, the average throughput for most experiments is reasonably high.

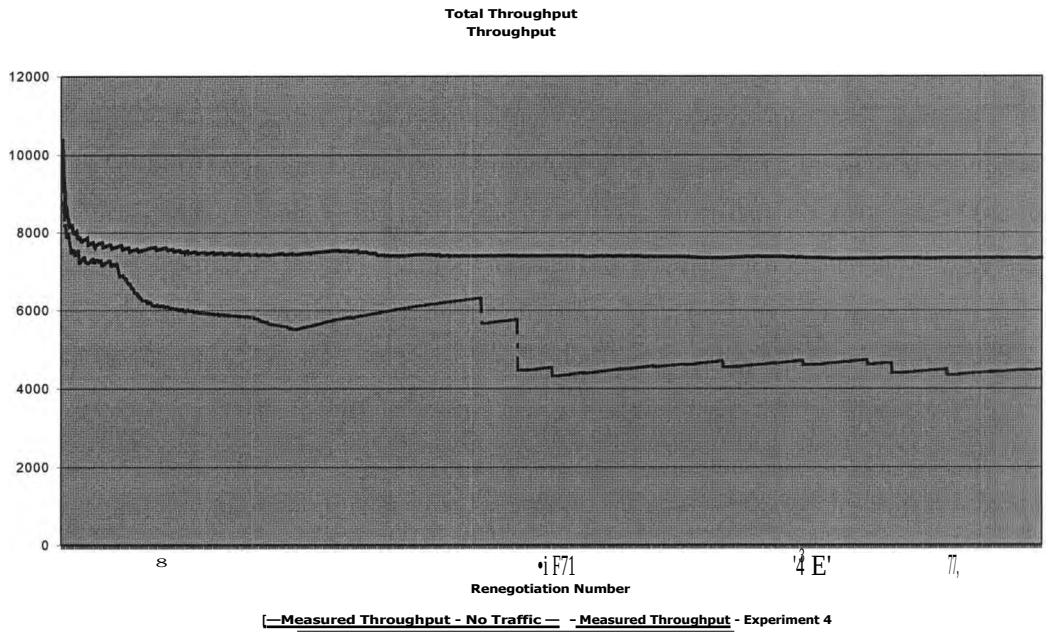

Figure 6.6. — Total Throughput method: Measured Throughput graph

Packet Loss

1:,xperiment Number	Average Packet Loss	Lost Packet Rao	Late Packet Ratio
o-Traffic	28.7 %	0.496	0.504
Experiment 1	15.5 %	0.719	0.281
Ex.eriment 2	16.0 %	0.591	0.409
Ex. eriment 3	44.6 %	0.735	0.265
Ex.eriment 4	49.9 %	0.669	0.331

As with other methods that are able to deliver a high average throughput, the *Total Throughput* method experiences an extremely high packet loss percentage. For example, Experiment 4 attempts to simulate real network traffic, and the measured loss is almost 50%. Although the measured loss for other experiments is lower than this value, average loss values between 30% and 50% are common.

Jitter

Experiment Number	Average Jitter	Average Inter-Packet Arrival Time
No-Traffic	53.063	149.16
Ex.eriment 1	104.872	233.739
Ex.eriment 2	43.569	320.767
Ex.eriment 3	126.861	212.61
Experiment.,.	153.203	243.53

Jitter is an excellent indication of the relationship between the delivered QoS and packet loss. The higher the average jitter, the larger the difference between the expected inter-packet arrival time and the actual inter-packet arrival time. Peaks in the jitter graphs correspond to the delivery of very late packets. As expected (due to the high average packet loss) the jitter values are extremely high.

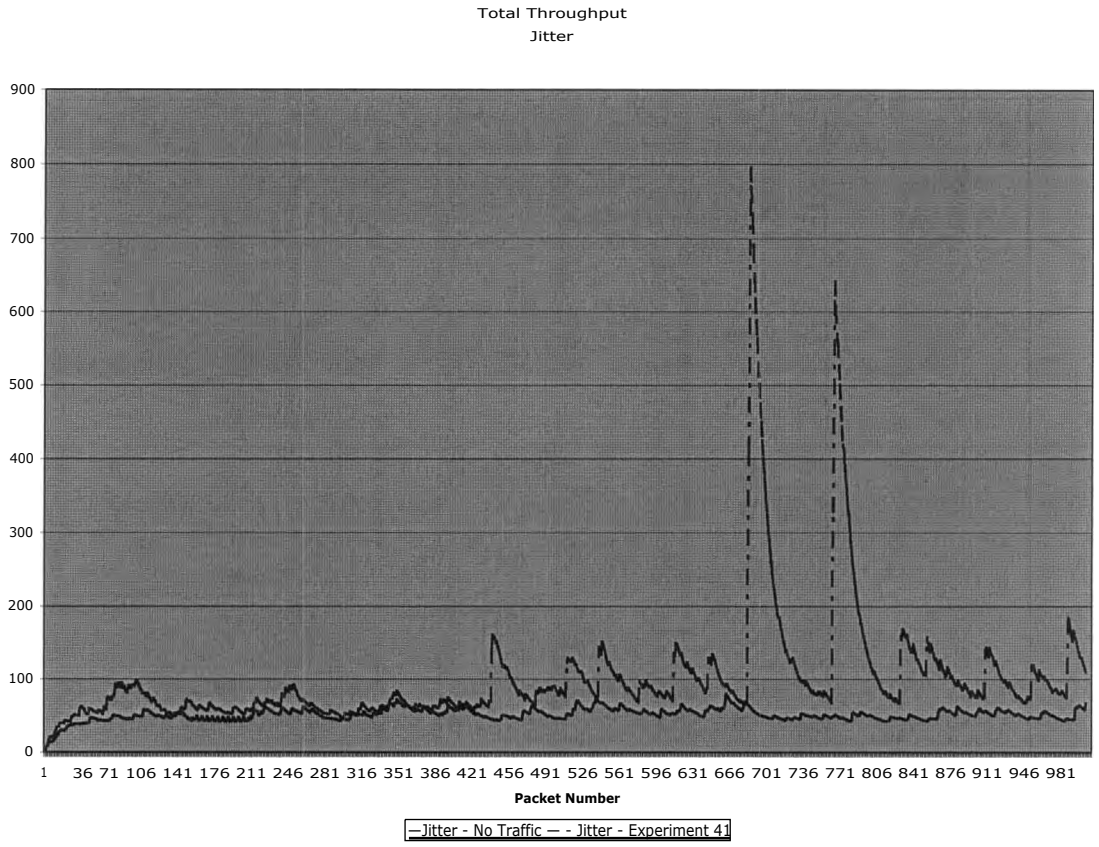

Figure 6.7. — Total Throughput method: Jitter graph

Conclusion

Experiment Number :	Maximum QoS Packet Count	Ratio	QoS Increase Ratio
No-Traffic	1	0.428	0.571
Ex eriment 1	1	0.858	0.141
Ex e eriment 2	1	0.222	0.778
Ex a eriment 3	1	0.984	0.016
Ex e eriment 4	1	0.909	0.091

The *Total Throughput* method is unable to rapidly decrease the QoS, except during the initial period of the connection. Increases in QoS are very rare after this point in time and the delivered QoS is maintained, regardless of the available resources. The *Total Throughput* method is excellent if a constant QoS is required with very few fluctuations in QoS. The high packet loss prevents the method from successfully delivering the requested QoS; thus this method is not ideal for use across heterogeneous IP networks.

6.3.2. Current Throughput Method

Throughput

Experiment Number	Average Throughput	Minimum Throu_ghput	Maximum Throughput
No-Traffic	5267.814	612.336	17062.5
Experiment 1	4976.527	602.206	8465.12
Experiment 2	4671.273	468.67	18827.6
Experiment 3	4606.219	- 638.596	8666.67
Experiment 4	4648.051	.630	8400
iiiiL_____AiiiiiiiiiiL			

As expected, the measured throughput fluctuates between the QoS that are easily deliverable across an unloaded network. The delivered QoS fluctuates mostly between 3600 Bps and 7200 Bps. Occasionally, however, a throughput of 7200 Bps is achieved and the delivered QoS is increased to 14400 Bps. Delivery of a constant throughput at 14400 Bps is not possible and high packet loss is observed. This leads to a lower measured throughput and a corresponding reduction in the delivered QoS. Extremely high throughput values (>15000 Bps) may occur in the following situation: two packets are sent from the sender to receiver, the first packet is extremely late (but arrives in the correct order) and the second packet arrives on time. The inter-packet arrival time between the first and second packet is therefore extremely small. The current throughput is calculated according to this value and will result in an extremely high throughput value being measured.

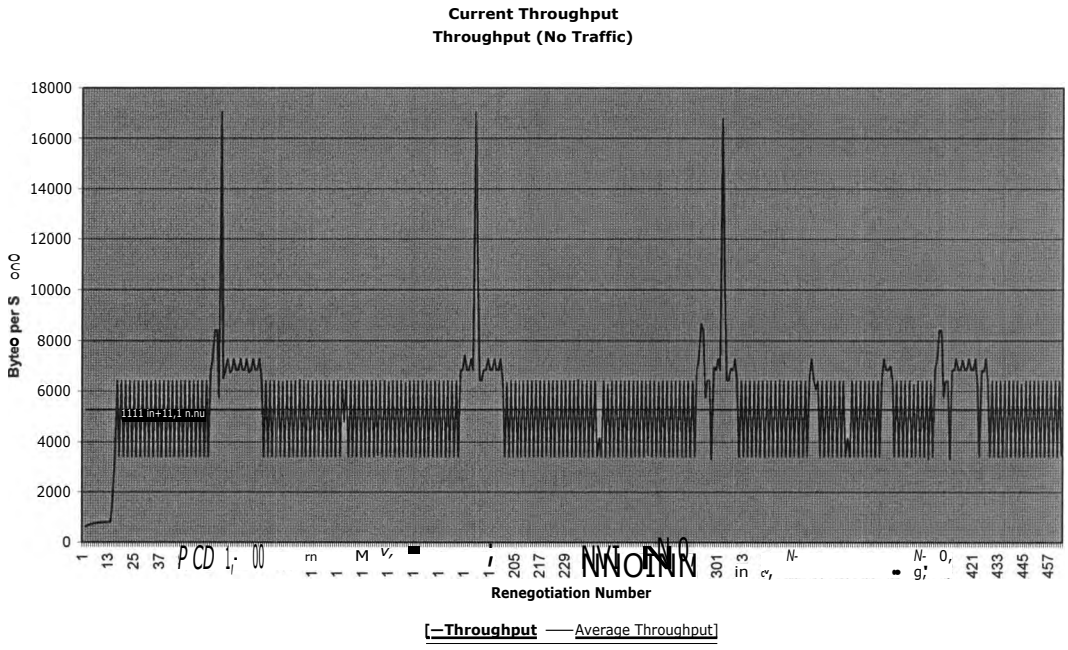

Figure 6.8. — Current Throughput method: Measured Throughput graph (No-Traffic)

Saturation of the line occurs in Experiments 1,2 and 4. The measured throughput is lower at the saturation points and the delivered QoS is reduced to the minimum value. After the saturation is complete, the measured throughput rapidly adjusts to fully utilise the available resources.

Experiment 3 delivers a constant, high level of network traffic, and the *Current Throughput* method is not able to adjust to the lower available resource level. The delivered QoS values are reasonably similar to the unloaded line values (No-Traffic Experiment), but the extra network usage results in a lower measured throughput when the delivered QoS is high. This is clearly evident by the lower average measured throughput value (No-Traffic: 5267 Bps; Experiment 3: 4606 Bps).

Experiment 4 is the most realistic indication of the renegotiation method's ability to react to "real" network traffic. The *Current* method performs a renegotiation for every received packet and is therefore able to adjust the delivered QoS very rapidly. This is clearly demonstrated by the fact that a QoS of 900 Bps (minimum possible QoS) is only delivered for one renegotiation period. The average throughput for Experiment 4 is 4648 Bps and this is very similar to the average throughput for the other experiments.

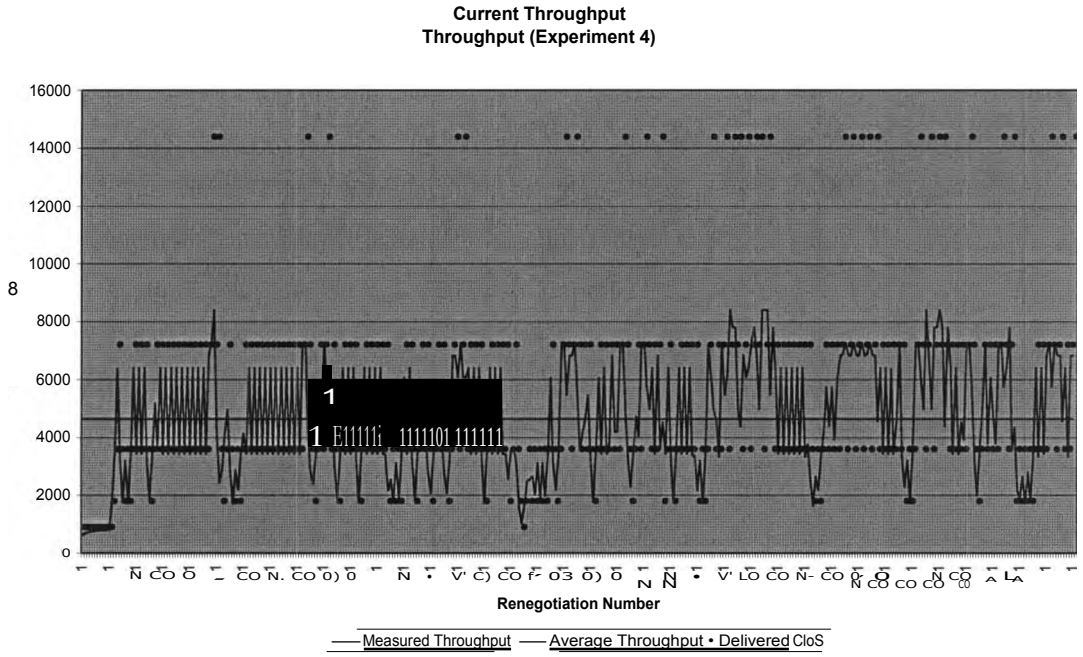

Figure 6.9. — Current Throughput method: Measured Throughput graph (Experiment 4)

Packet Loss

Experiment Number	A verage Packet Loss	Lost Packet Ratio	Late Packet Ratio
No-Traffic	9.9 %	0	1.000
Experiment **1**	28.0 %	0.455	0.545
Experiment 2	23.0 °A	0.458	0.542
Experiment 3	28.0 %	0.421	0.579
Experiment 4	16.9 %	0.162	0.838

Although packet loss does not directly affect the renegotiation of the QoS, it is an important value to measure in order to determine the effectiveness of the renegotiation policy. If the delivered QoS is too high, then a large packet loss will occur. Although lost and out-of-order packets are not included in the throughput calculation, late packets are included. Therefore, renegotiation methods with a high late packet ratio (but a low lost packet ratio) will still achieve a high throughput. Across an unloaded line, the measured loss is approximately 10%, which is determined during delivery of the maximum QoS (14400 Bps).

Due to the correlation between a high QoS delivered and a large loss measured, experiments that attempt to transfer the maximum QoS frequently experience the largest increase in average loss. Experiments 1 and 3 both encounter an average packet loss of nearly 30%. Packet loss in Experiment 4, however, increased to only 17% while still managing to achieve a reasonably high throughput. Most packet loss (for all experiments) occurs during periods of high delivered QoS rather than due to the level of saturation on the network. This illustrates the ability of the renegotiation method to rapidly adjust to an increase in network traffic and deliver a lower QoS.

Jitter

Experiment Number	Average Jitter	Average Inter-Packet Arrival Time
No-Traffic	15.435	238.218
Experiment 1	64.019	228.9
Experiment 2 -a	44.817	281.816
Ex • erirnent 3	69.134	244.053
Experiment 4	46.572	264.073
· · ·,_·· · ·,=		

Fluctuations in the measured jitter are directly related to the delivered QoS. Across an unloaded network, a QoS of 900 Bps — 7200 Bps is easily deliverable and therefore a jitter of close to zero is achieved. However, when the delivered QoS is increased to 14400 Bps, the number of late packets

increases dramatically and a corresponding increase in the jitter value is evident. The average jitter value for all experiments is between 44 and 70. This illustrates the fact that regardless of network usage, the *Current Throughput* method attempts to deliver a QoS that uses the available resources.

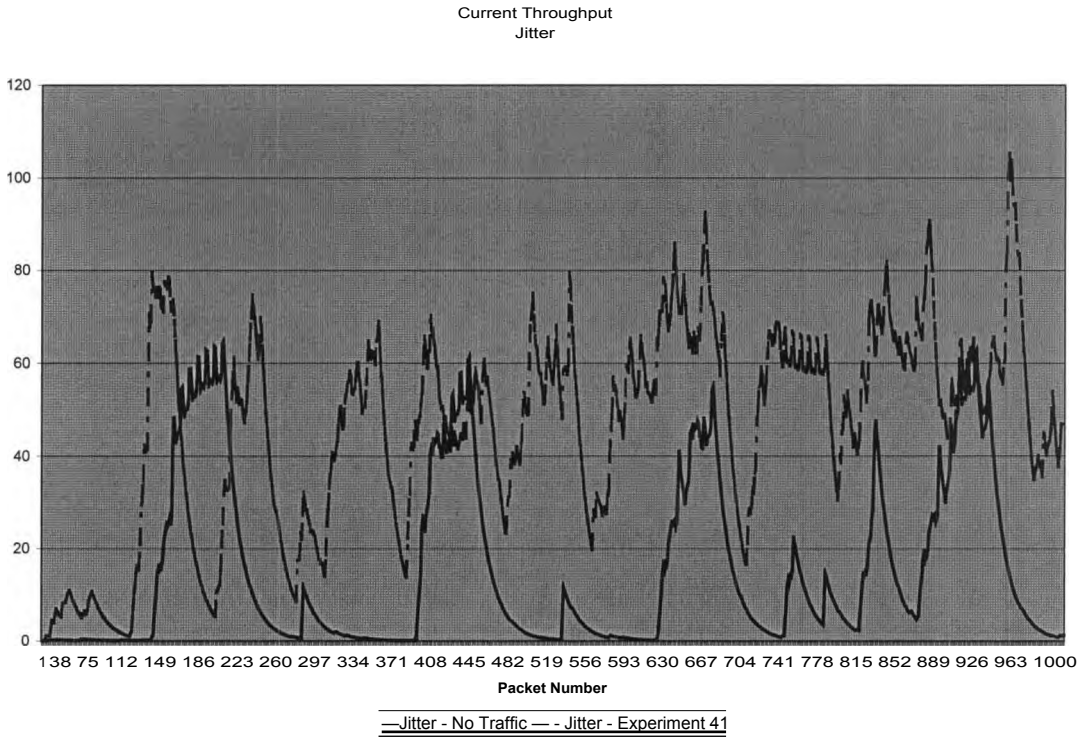

Figure 6.10. — Current Throughput method: Jitter graph

Conclusion

Experiment Number	Maximum QoS Packet Count	QoS Decrease Ratio	QoS Increase Ratio
No-Traffic	109	0.489	0.510
Ex e eriment 1	51	0.477	0.523
Ex eriment 2	129	0.477	0.523
Ex e eriment 3	31	0.480	0.520
Ex e eriment 4	88	.479	0.521

The major advantage of this method is the ability to rapidly adjust to changing network conditions. Although the QoS is enlarged by only a single QoS group per renegotiation, the renegotiation period is a single data packet, thereby increasing the QoS reasonably rapidly. If a decrease in the delivered QoS is required, it is decreased to the measured throughput and during periods of high network usage rapid decreases are experienced. This rapid decrease in QoS ensures that a low packet loss occurs, regardless of the experiment.

Although this method is able to adjust extremely quickly to fluctuations and still maintain a low packet loss ratio, a major disadvantage is the fluctuations in delivered QoS. The QoS is adjusted on every renegotiation and is therefore extremely annoying to the user. The *Current Throughput* method is able to adjust well if there is either a large or small amount of network traffic but is not able to maintain an acceptable QoS during periods of medium activity.

6.3.3. Smoothed Throughput Method

Throughput

Experiment Number	Average Throughput	Minimum Throughput	Maximum Throughput
No-Traffic	6990.854	3412.5	8205.16
Ex . eriment 1	5005.018	1904.65	7802.53
'Ex . eriment 2	4960.715	887.805	7874.14
Experiment 3	5281.467	2247.57	7184.21
Ex • eriment 4	5537.282	3254.3	7749.68
i11			

Throughout all experiments, the smoothing fluctuation was set at a value of 2 QoS groups. This allowed the throughput to increase or decrease by a factor of 4. Across an unloaded line, the delivered QoS generally fluctuates between the 14400 Bps and 7200 Bps. The reason why the decrease in delivered QoS is generally only one group and the smoothing fluctuation is 2 QoS groups, is because the delivered QoS is decreased to the closest QoS group that is lower than the measured throughput. The delivered QoS is only reduced to a value of 3600 Bps four times out of 35 renegotiations performed. After delivering a QoS of 3600 Bps for the entire renegotiation period, a maximum increase to 14400 Bps is performed. This results in an extremely high throughput (6990 Bps) for the experiment.

Network saturation has a large effect on measured throughput. After reducing the throughput sufficiently to accommodate the extra traffic, the delivered QoS is again increased to maximum due to the smoothing fluctuation value. The delivered QoS oscillates between 3600 Bps and 14400 Bps for most of the experiments and this results in a high average throughput for all experiments. Occasionally, a value of 7200, 1800 or 900 Bps is delivered but these are extremely rare.

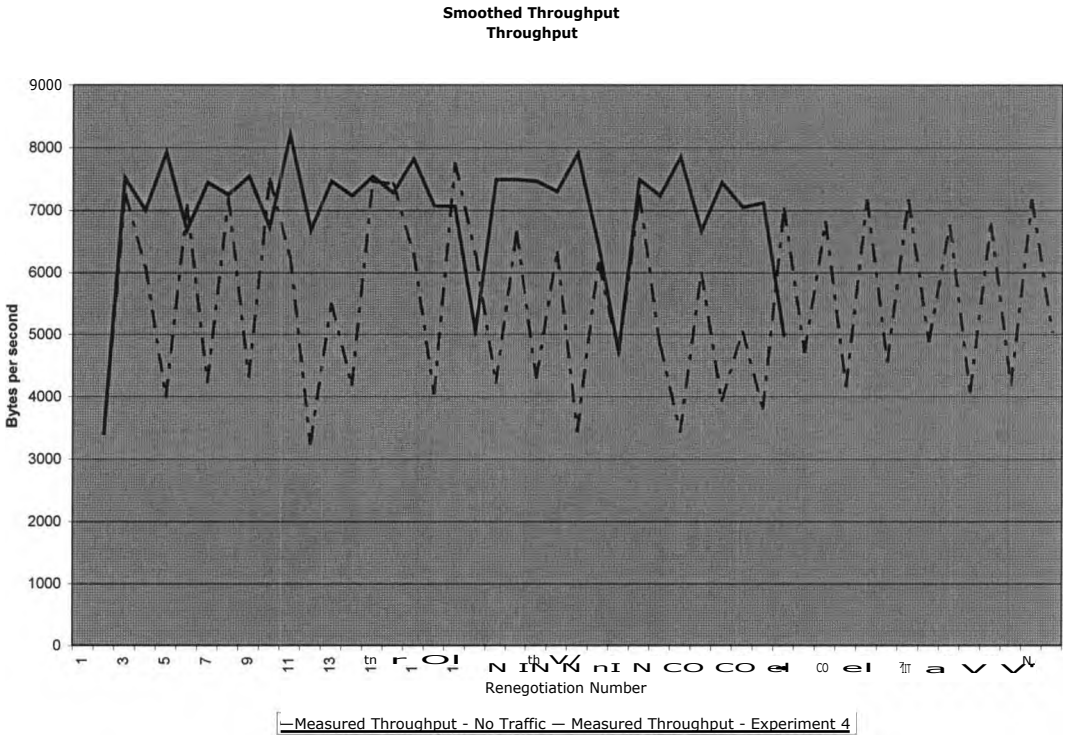

Figure 6.11. — Smoothed Throughput method: Measured Throughput graph

Packet Loss

Experiment Number	Average Packet Loss	Lost Packet Ratio	Late Packet Ratio
No-Traffic	35.0 %	0.611	0.389
Ex [illegible]	47.2 %	0.659	0.340
Ex [illegible]	49.2 %	0.706	0.294
[illegible]	41.7 %	0.564	0.436
[illegible]	40.3 %	0.573	0.427
[illegible]		0 PORMIRPRIPI	

Although important, a high average throughput is not sufficient for an experiment to be considered a success. The packet loss incurred while delivering the throughput is also important. The delivery of a throughput of 14400 Bps for nearly half of all renegotiation periods (in all experiments) resulted in an unacceptable packet loss occurring. Across an unloaded line the packet loss was 35%, mostly whilst delivering a QoS of 14400 Bps. Due to the low number of QoS groups (5) and the high smoothing fluctuation (2), the QoS was increased from an acceptable value to an unacceptable value too quickly. Except during periods of network saturation or high utilisation, a throughput of 7200 Bps could be maintained. If a QoS of 3600 Bps was successfully delivered, however, the QoS was immediately

increased to 14400 Bps and large packet loss would occur. The packet loss for all experiments is between 35% and 50% and this value is too high for the method to be considered successful, regardless of the high throughput achieved.

Jitter

Experiment Number	1verage Jitter	Average Inter-Packet Arrival Time	Packet Arrival Rate
No-Traffic	57.293	160.458	6.082
Ex	100.100	220.061	4.377
Ex	93.730	224.894	4.434
Ex	81.857	204.989	4.671
Ex	75.266		4.886

As already discussed, jitter is related to the number of late packets delivered. The later the packet arrives, the higher the measured jitter. Therefore, as expected, the *Smoothed Throughput* method suffers from an extremely high average jitter value (57.3 for the No-Traffic Experiment). Late packets mainly occur during delivery of the maximum QoS and a large percentage of delivery of this QoS is responsible for a high jitter value. All data packets that are received in the correct order are included in the jitter calculation, even packets that are too late to be output. Very late packets can therefore effect the overall jitter value.

Jitter values for all experiments were extremely high (75 —100). If the throughput is too high, the jitter value increases due to the late and lost packets. If jitter is reasonably low, the delivered QoS is able to be comfortably delivered, in-time and with a low packet loss. Jitter is therefore a good indication of throughput versus packet loss.

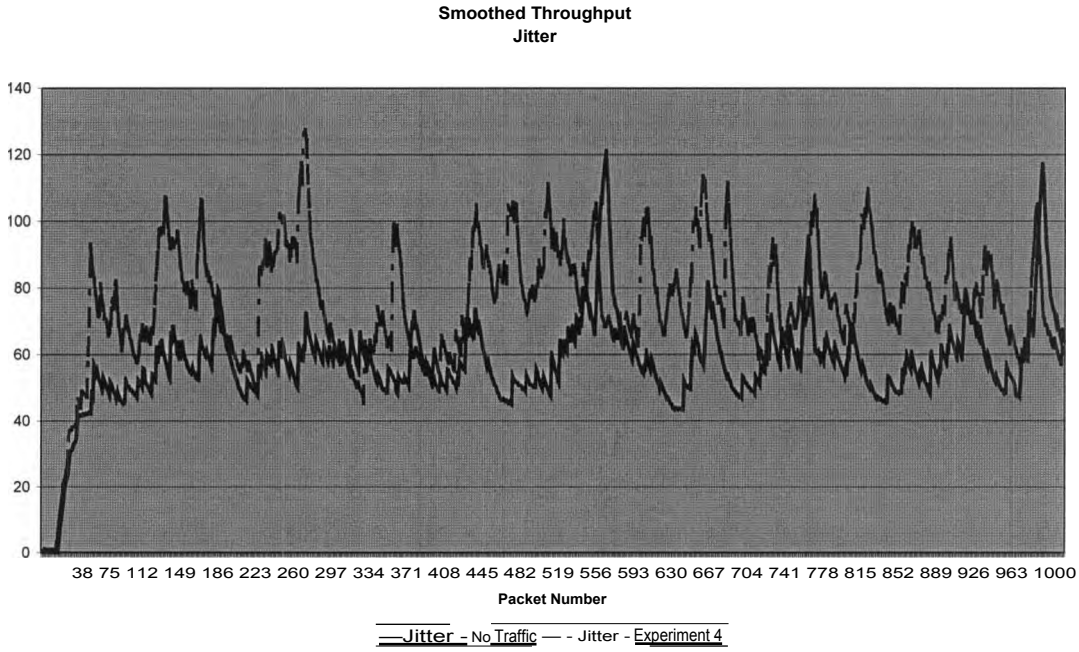

Figure 6.12. — Smoothed Throughput method: Jitter graph

Conclusion

Experiment Number	Maximum QoS Packet Count	QoS Decrease Ratio	QoS Increase Ratio
No-Traffic	16	0.500	0.500
*Experiment 1	16	0.526	0.473
Experiment 2	15	0.538	0.462
Experiment 3	15	0.500	0.500
kifitent 4	15	0.530	0.470

The use of a smoothing fluctuation value to increase or decrease the delivered QoS rapidly, and therefore utilise all available resources, is a good idea. Unfortunately, the smoothing fluctuation led to the QoS being constantly delivered at a level that was too high for the network to successfully handle. Although a high average throughput was delivered, a correspondingly high packet loss rendered this method unsuccessful.

With only 5 QoS groups, a maximum allowed fluctuation value of 1 group would be more ideal. This would prevent the QoS from continually fluctuating between 3600 Bps and 14400 Bps. If a smoothing fluctuation of 2 is required, it is possible to insert extra QoS groups into the QoS table and a slower increase in the delivered QoS would occur. A smoothing fluctuation of 2 is, however, ideal for

decreasing QoS. A rapid reduction in throughput during saturation periods ensured minimum loss during these times.

The *Smoothed Throughput* method can also be improved by not including late packets in the throughput calculation. This will result in a lower measured throughput and a correspondingly lower packet loss. With a smoothing fluctuation of 2, the frequency of the maximum delivered QoS will remain high and a high packet loss will result.

6.3.4. Congested Throughput Method

Throughput

Experiment Number	Average **Throughput**	Minimum **Throughput**	i\laximum **Throughput**
No-Traffic	4[7]45.547	890.459	5040
Experiment **1**	5262.096	, - 1944.21	7469.71
Experiment 2	4710.083	'-— 892.643	8181.11
:Experiment 3	5265.814	47.8737	7443.04
– E**a,eximent** 4	5380.519	. „„t187 805	7518.69

The measured throughput determines a network "state" which in turn determines the maximum fluctuation allowed for the next renegotiation period. For all experiments, the following states were defined:

State	Throughput	Fluctuation
Unloaded	>= 7400 B s	1111.1111r
	300 > x < 7400	11111Elffii
ested	<= 300 B s	

Figure 6.13. - Network-state and allowed fluctuation for the Congested Throughput method

The measured throughput (4245 Bps) across an unloaded network was surprisingly low compared to the *Smoothed* method (6990 Bps). The delivered QoS values for both methods were reasonably similar, as well as the measured packet loss. The maximum throughput for the No-Traffic Experiment was only 5040 Bps, which was also surprising. Due to the extremely low defined congested value, the network-state never became congested and fluctuated continuously between the unloaded and loaded states for all experiments.

The average throughput for all experiments was extremely similar to the *Smoothed Throughput* method. This was due to the definition of the network-state and the corresponding fluctuations. The network was in the loaded state for most of the experiments, and had a fluctuation of 2. This is the same as the defined fluctuation for the *Smoothed Throughput* method. Due to the similar fluctuation value, the delivered QoS were comparable, and therefore, comparison of the throughput values could be made.

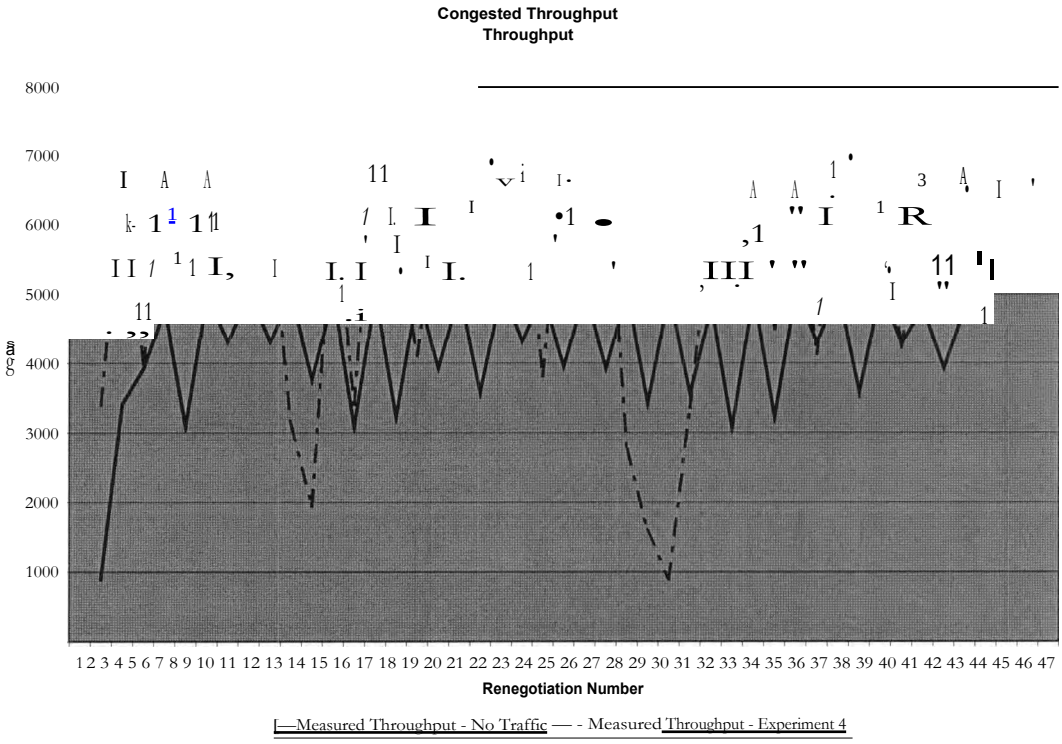

Figure 6.14. — Congested Throughput method: Measured Throughput graph

Packet Loss

Experiment Number 3111111111	Average Packet Loss	Lost Packet Ratio	Late Packet Ratio
No-Traffic	34.4 %	0.590	0.410
Experiment 1	42.9 %	0.569	0.431
Experiment 2	36.0 %	0.650	0.350
Experiment 3	41.9 %	0.603	0.397
Experiment 4	40.7 %	0.602	0.398

Packet loss was slightly lower than the *Smoothed Throughput* method but still extremely high. The difference is that packet loss is caused by the *Congested* method having a slightly lower overall average throughput.

Jitter

Experiment Number	Average Jitter	Average Inter-Packet Arrival Time	Packet Arrival Rate
No-Traffic	66.804	176.430	6.106
	80.715	205.334	4.655
	63.024	245.304	4.222
	84.262	228.4	4.661
	68.740	205.1	4.749

As already discussed, jitter is related to the delivered QoS. The delivered QoS for the *Smoothed* and *Congested* methods are extremely similar and it follows that a corresponding relationship will occur in the jitter calculation results (between 57 and 100 for the *Smoothed* method, and between 63 and 84 for the *Congested* method)

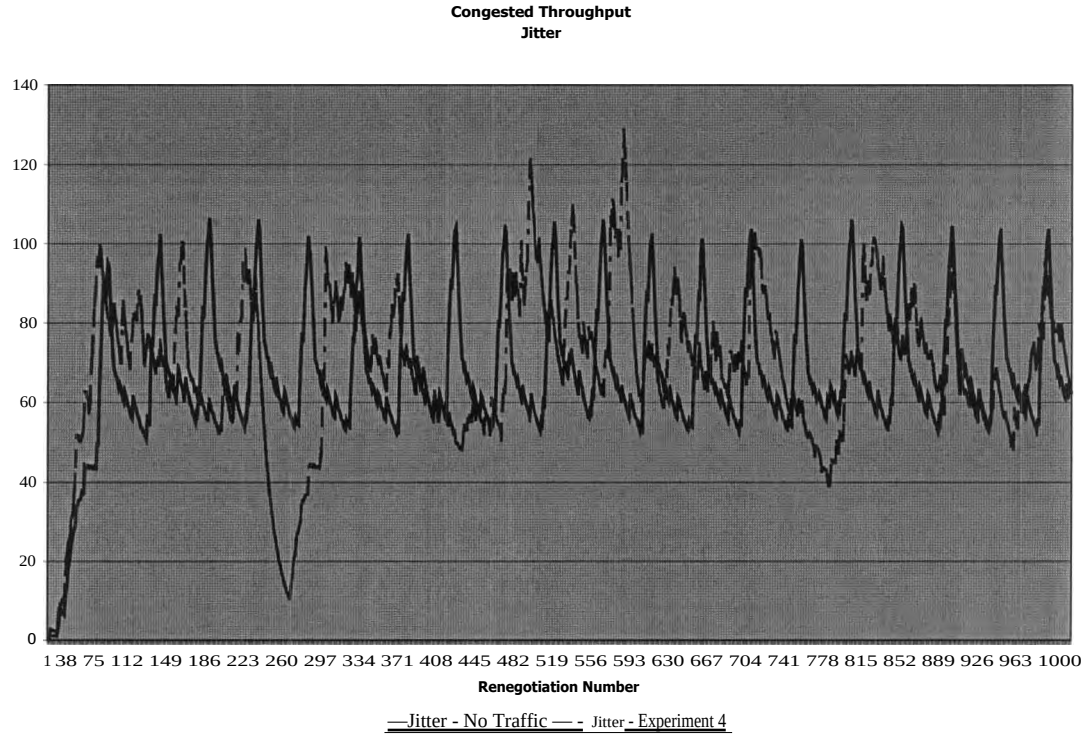

Figure 6.15. — Congested Throughput method: Jitter graph

Conclusion

Experiment Number	Maximum QoS Packet Count	QoS Decrease Ratio	QoS Increase Ratio
No-Traffic	16	0.500	0.500
Experiment 1	16	0.490	0.510
Experiment 2	14	0.568	0.432
Experiment 3	15	0.50	0.491
Experiment 4	16	0.541	0.459

Although the introduction of a network-state was intended to differentiate between the *Smoothed* and *Congested* methods, the state variables were set at levels that prevented any differences from being distinguished. For example, if the delivered QoS is 14400 Bps and the measured throughput during this renegotiation period is greater than 7400 Bps, the network state is set to unloaded and a maximum fluctuation of one is permitted. If saturation of the network now occurs, a rapid decrease of the delivered QoS is prevented, and a QoS that is too high, is delivered.

If, however, the fluctuation were 4, then during the network saturation period the QoS would have been reduced to the minimum possible value. In the unused network situation, the QoS would be reduced to the QoS group closest to the measured throughput, also 7200 Bps. The fluctuation value therefore prevented rapid decrease, and demonstrated that the same problems exist with the *Congested* and the *Smoothed* methods.

The state variables are easier to define with a larger range of delivered QoS. The maximum throughput across a PPP line is theoretically 14400 Bps. Experimentation with full CD quality audio would permit a range of state values from 11025 Bps to 176400 Bps. A fairer indication of the smoothing fluctuations could then be measured.

6.3.5. Throughput Renegotiation Method Conclusions

A high average throughput is measured for most methods based on throughput. This is only possible, however, due to an extremely high packet loss. Late packets are currently included in the throughput calculation. A decrease in throughput (and an indirect reduction in the measured packet loss) may be achieved by not including these packets in the calculation.

A comparison between the loss and throughput methods illustrates the need to maintain the QoS at renegotiation time. During the experiments, it was observed that a measured throughput just outside

the acceptable window could be delivered with a low packet loss. The acceptable window must be set accurately in order to prevent high packet loss, but allowing the highest possible throughput.

A reduction in the jitter value is possible in two ways, namely: the maximum possible QoS is not delivered as frequently, and late packets are not included in the calculation. Late packets are currently included in all jitter calculations. Delivery of a lower QoS is possible by reducing the acceptable window for the increase in throughput.

Currently, the *Smoothed* and *Congested* methods are not sufficiently different, and as a consequence the experiments produced similar results. This similarity is also due to the setting of the threshold values in the *Congested* method. Although the range of possible throughput values is small for the PPP line (14400 Bps), the setting of accurate state values is more important. The *Congested* method failed to utilise the network-state mechanism sufficiently.

Due to the small number of QoS groups (5) a fluctuation of 2 allowed for rapid increase and decrease in delivered QoS. As can be seen by the large number of lost packets, the delivered QoS increased too rapidly. A maximum increase of only one QoS group should have been allowed, regardless of the network-state.

Expanding the number of QoS groups would also reduce the rate at which the delivered QoS is currently increased. Due to the existing method of doubling the delivered QoS for each increase in QoS group, a larger throughput range is required.

6.4. Loss Renegotiation Methods

In this section, an analysis of all QoS renegotiation methods based on the measured packet loss is investigated. An in-depth analysis of only the *Smoothed* and *Congested* methods is provided, this is because it is not possible to calculate the packet loss per received packet and therefore the *Current* method is not discussed. The *Total* method produced similar results for all experiments, and only a brief analysis of these results is discussed. The *Smoothed* and *Congested Packet Loss* methods are discussed in detail and the results discussed in the same manner as the throughput renegotiation methods. Jitter is not discussed in detail due to the similarity in results to the throughput methods already discussed.

6.4.1. Total Packet Loss Method

Experiment Number	Average Throughput	Minimum Throughput	Maximum Throughput
No-Traffic	945.469	89 .845 -	1183.74
Experiment 1	946.406	., 896.354	1183.74
Experiment 2	943.942	896.465	1183.74
Experiment 3	981.409	896.495 ,	1299.63
Experiment 4	920.165	4..,::.4-495.4g _	1183.74

Experiment Number	Average Packet Loss	Packet Ratio	Late Packet Ratio
No-Traffic	6.8 %	0.561	0.439
	3.6 %	0.574	0.425
	2.5 %	0.0	1.000
	1.9 %	0.450	0.550
111MMEM	9.6 %	0.650	0.350

Experiment Number	Average Jitter	Average Inter-Packet Arrival Time
No-Traffic -	8.977	1091.571
Experiment 1	21.739	893.154
Experiment 2	11.547	1116.548
Experiment 3	13.587	1201.298
Experiment 4	14.118	1074.692

Experiment Number	Maximum QoS Packet Count	QoS Decrease Ratio	QoS Increase Ratio
* No-Traffic	57	0.952	0.048
Experiment 1	353	0.966	0.033
Experiment 2 *	57	0.826	0.174
Experiment 3	57	0.970	0.030
Experiment 4	57	0.956	0.044

Unlike the *Total Throughput* method, it was not necessary to start the experiments with a maximum initial QoS. The first packet received at the client, regardless of initial QoS, was normally late and the total packet loss statistic therefore began with a 100% percent loss. This high initial packet loss caused the delivered QoS to decrease rapidly and in turn caused a corresponding decrease in measured throughput. The packet loss was continuously reduced due to the low delivered QoS and was eventually within the minimum acceptable packet loss value. At this point, the QoS rapidly increased and a high throughput

and packet loss was observed. The high packet loss then caused the QoS to rapidly decrease to the lowest possible value. This process continually repeated itself, regardless of network traffic.

This slow decrease in QoS, followed by rapid increase, was observed during all experiments. The *Total Packet Loss* method therefore reacts almost identically regardless of network load. This method is of no further interest and will not be discussed again.

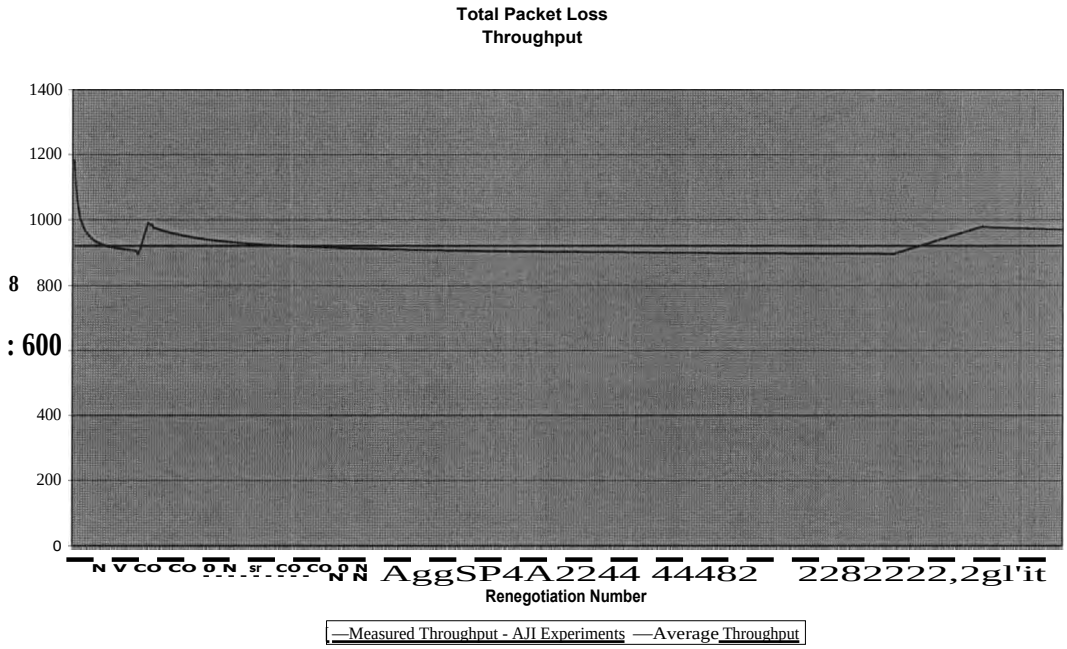

Figure 6.16. — Total Packet Loss method: Measured Throughput graph

6.4.2. Smoothed Packet Loss Method

Throughput

Experiment Number	Average Throughput	Minimum Throughput	Throughput
No-Traf . c	5082.558	866.896	7545.9
Ex eriment 1	4410.259	776.303	7518.69
Ex eriment 2	4121.562	880.645	7518.69
	4227.323	880.645	6933.33
	4156.005	880.645	7543.42

Renegotiation based on packet loss produced lower measured throughput values for the No-Traffic Experiment (5082 Bps), compared to a measured throughput of 6990 Bps for the *Smoothed Throughput* method. Measured throughput for the experiments with network traffic were relatively closer to the baseline results than the throughput methods.

During network saturation periods, the measured throughput is reduced to 900 Bps and the delivered QoS is decreased rapidly during this period. After successfully delivering this QoS, the smoothing fluctuation causes the delivered value to be increased by a factor of 4 and high throughput is then measured during this period.

During non-saturation periods the large fluctuation value results in very similar measured throughput for all experiments. A QoS of 3600 Bps can be delivered with a low packet loss regardless of network activity. The delivered QoS is then increased to the maximum value. The delivered QoS is therefore extremely similar to the *Smoothed Throughput* method, and similar results were experienced.

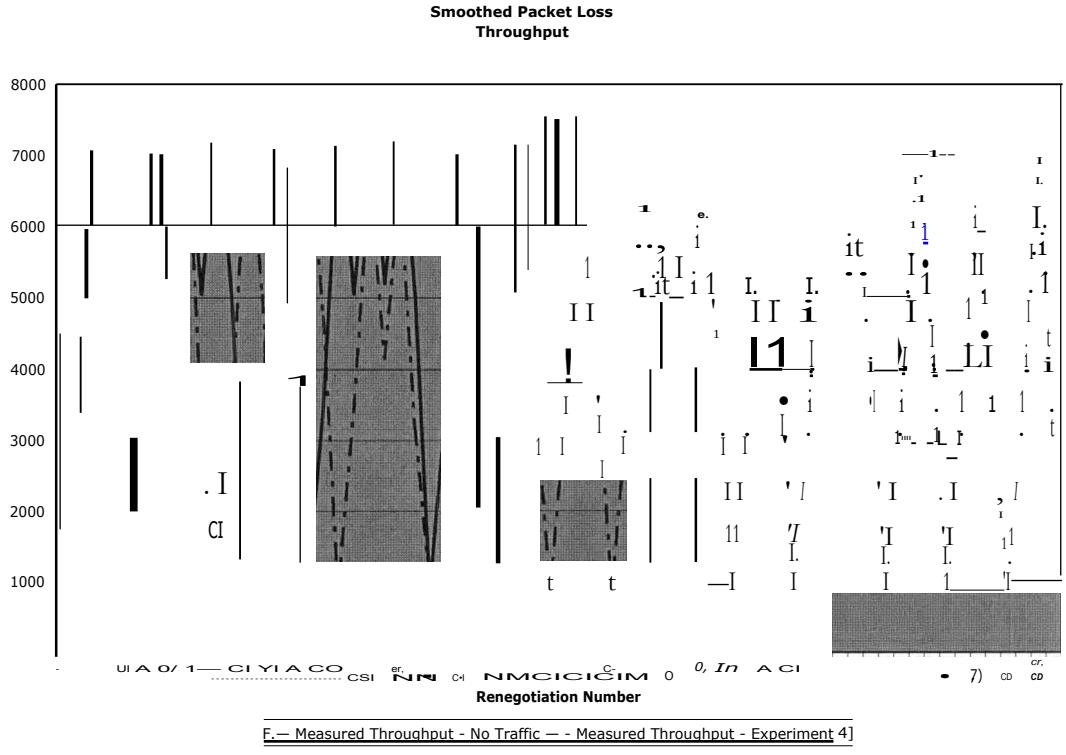

Figure 6.17. — Smoothed Packet Loss method: Measured Throughput graph

Packet Loss

l'iveriment Number	Average Packet Loss	Lost Packet Ratio	Late Packet Ratio
No-Traffic	27.2 °A)	0.256	0.744
`Ex ·4	34.5 %	0.459	0.540
Ex eriment 2	36.2 %	0.553	0.446
Ex. .... •	37.5 %	0.560	0.440
Ex eriment 4	37.4 %	0.552	0.447

During the renegotiation process, measured packet loss determines whether the QoS is increased or decreased. For all experiments the following packet loss limits were used:

Renegotiation Performed	Packet Loss
Increase •oS	< 0.02
Maintain • oS	0.10 > x > 0.02
Decrease QoS	> 0.10

Figure 6.18. - Renegotiation threshold values for the Smoothed Packet Loss method

Packet loss for all experiments is much lower than the corresponding *Smoothed Throughput* experiments. Although this is encouraging, the loss is still far too high and the received audio stream is noticeably broken. As with all other methods, the largest percentage of packet loss is experienced during the maximum QoS delivery periods.

Increases in the QoS for both the *Smoothed Throughput* and *Smoothed Packet Loss* methods are controlled by the smoothing fluctuation. As discussed in section 6.3.3, this value causes the delivered QoS to be increased too rapidly and large packet loss then occurs.

Conclusion

Experiment Number	Average Jitter	Average Inter-Packet Arrival Time	Packet Arrival Rate
No-Traffic	60.377	218.274	4.502
Experiment 1	71.279	250.24	3.958
Experiment 2	70.658	276.815	3.588
Experiment 3	74.133	259.95	3.844
Ex • eriment 4	65.994	291.139	3.466

Experiment Number	Maximum QoS Packet Count	QoS Decrease Ratio	QoS Increase Ratio
o-Traffic	16	0.490	0.510
Ex eriment 1	16	0.491	0.509
Ex eriment 2	14	0.500	0.500
Ex eriment 3	16	0.500	0.500
Ex eriment ,	15	0.500	0.500

Packet loss is an important aspect of QoS. High throughput is never acceptable if a correspondingly high packet loss is experienced. Although the *Smoothed Packet Loss* method is able to produce a lower packet loss than the throughput methods, the measured loss is still too high for an acceptable QoS to be received.

The disadvantages of this method are the same as already discussed for the *Smoothed Throughput* method - the smoothing fluctuation causes the delivered QoS to increase too quickly and results in large packet loss. Increases in QoS must be limited to a single QoS group and renegotiation performed on the newly measured value. If a QoS of 7200 Bps can be delivered with less than 2% packet loss, then it is possible for the QoS to be increased to 14400 Bps. During periods of moderate network activity this is unlikely, and the QoS is able to remain at an acceptable level.

6.4.3. Congested Packet Loss Method

Throughput

Experiment Number	Average Throughput	Minimum Throughput	Maximum Throughput
No-Traffic	5307.616	1040	7543.42
Experiment 1	4796.53	102.535	7518.69
Experiment 2	4392.787	876.941	8181.11
Experiment 3	4958.081	883.019	6893.02
Experiment 4	4886.482	880.645	7518.69

As is to be expected, the results for the *Congested Packet Loss* method are similar to a combination of the *Smoothed Packet Loss* and the *Congested Throughput* methods. The average throughput is lower than the value measured for the *Congested Throughput* method. The *Congested* method is able to react to fluctuations in QoS more successfully than the *Smoothed Packet Loss* method due to the higher percentage of delivery of a maximum QoS. Although the average measured throughput for all experiments is reasonably high, as already discussed for other methods, this value is a product of the high percentage of delivered QoS, which results in a large packet loss. The measured throughput value alone is therefore not a good indication of the success of a method.

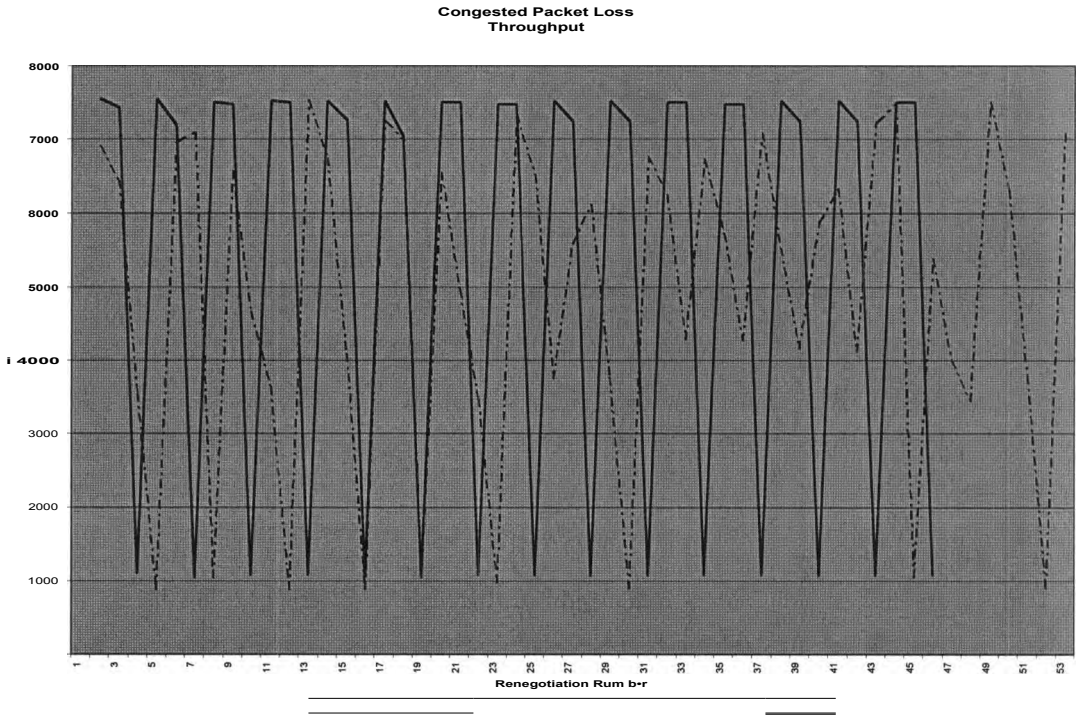

Figure 6.19. — Congested Packet Loss method: Measured Throughput graph

Packet Loss

Experiment Number	Average Packet Loss	Lost Packet Ratio	Late Packet Ratio
No-Traffic	24.6 %	0.278 11	0.722
Experiment 1	37.1 %	0.571	0.428
Experiment 2	39.6 %	0.651	0.349
' Experiment 3	38.3 %	0.624	0.376
Ex eriment 4	37.1 %	0.589	0.411

Discussion of the connection between the delivered QoS and packet loss has already been entered into and is not necessary to revise for this method as well. An illustration of how the incorrect state and fluctuation values are used is, however, necessary. In Experiment 3, the measured throughput was never higher than 7200 Bps. If a QoS of 900 Bps was delivered with less than 2% packet loss, the state became congested and a fluctuation of 4 caused the delivered QoS to increase to the maximum. Large packet loss then occurred and the QoS was reduced immediately.

Conclusion

1:xpertment Number	Average Jitter	**Average Inter-Packet Arrival Time**	Packet Arrival Rate
No-Traffic	57.721	220.958	4.696
Experiment **1**	70.140	233.329	4.456
Ex eriment 2	69.475	256.036	3.895
Experiment 3	67.306	229.279	4.308
Experiment 4	65.932	231.276	4.324
AIL,		_	

	Maximum		
No-Traffic	5	0.500	0.500
Experiment 1	4	0.612	0.388
Experiment 2	4	0.615	0.385
Experiment 3	4	0.641	0.359
Ex. eriment 4	5	0.600	0.400

Both the smoothing fluctuation and network-state variables must be set at the correct levels in order for all the methods using these values, including this experiment, to be successful. For the *Smoothed Packet Loss* method, the values are clearly not set correctly. Large fluctuations should be allowed when the

state is unloaded and only small fluctuations permitted while in the congested state. Increases in QoS must be restricted to a single group, while decreases are allowed up to the defined smoothing fluctuation value.

6.4.4. Loss Renegotiation Methods Conclusions

Loss methods exhibit similar characteristics to the corresponding throughput methods. Currently, the two method groups (loss and throughput) are not sufficiently different for either method to be recommended above the other one.

All the suggested changes for the throughput methods apply to the loss methods as well: not including late packets in throughput calculation; the changing of state values for the *Congested* method, and finally, changing the fluctuation values to allow for rapid decrease but slow increase in the delivered QoS.

Throughout all experiments, it is clearly evident that loss and throughput are directly related. The higher the delivered QoS (assuming an unused network), the larger the measured packet loss. Renegotiation should therefore not depend solely on either loss or throughput, but a combination of both calculations should be used instead. The *Adjusted* method to be discussed in section 6.5 attempts to combine all the advantages of the previous methods whilst removing the described deficiencies.

6.5. Adjusted Method

Throughput

Experiment Number	Average Throughput	Minimum Throughput	Maximum Throughput
No-Traffic	6287.588	1760.62	6453.05
Ex • se	4746.884	883.019	7186.57
Ex ‖ oo 9	4492.976	880.645	6455.17
„- Ex eriment 3	4031.072	887.805	6046.91
x eriaient 4	4457.598	,.., ,883.M5.:	7724.76

The *Adjusted* method attempts to deliver the highest possible QoS while maintaining an "acceptable" packet loss. Across an unloaded line, the delivered QoS is maintained at a constant 7200 Bps and this is achieved with a measured packet loss of almost 0%. Unlike the throughput methods, the *Adjusted* method is able to deliver a constant QoS as long as the measured values do not require any

renegotiation. The average throughput is 6287 Bps and this value is slightly outside the acceptable window for an increase in the delivered QoS (6480 Bps).

In order to prevent the delivered QoS from constantly returning to a maximum value, increases in QoS are restricted to a single QoS group. Decreases, however, are determined by the smoothing fluctuation for the current state. The allowed rapid decrease ensures that saturation of the network does not result in a high delivered QoS. If the minimum QoS is comfortably delivered, the QoS increases by only a single group. If the network is still saturated, this prevents a large packet loss from occurring and the minimum QoS is then re-delivered at the next renegotiation point.

Experiment 3, delivery of half-saturating network traffic, was the least successful experiment. This is due to the fact that although a QoS of 3600 Bps is easily deliverable, 7200 Bps resulted in high packet loss. Addition of extra QoS groups to the QoS table would prevent this from occurring. Experiment 4 offers a clear indication of the success in altering the smoothing fluctuation that is dependent on the network-state. At the point that network saturation occurs, the delivered QoS is 14400 Bps (unloaded state, maximum fluctuation of 4). The delivered QoS is immediately reduced to 900 Bps, and this is maintained for the following period. After the saturation is complete, the QoS slowly increases to a level that is comfortably delivered.

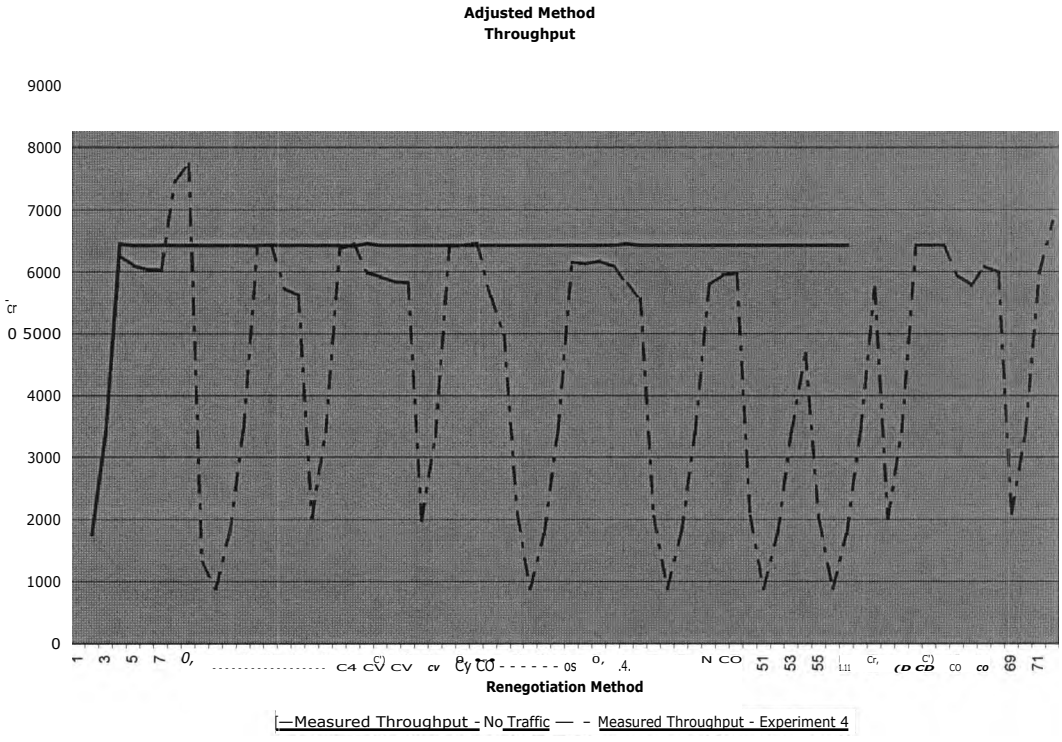

Figure 6.20. - Adjusted method: Measured Throughput graph

Packet Loss

Experiment Number	.1verage Packet Loss	Lost Packet Ratio	Late Packet Ratio
No-Traffic	0.8 %		1.000
Ex erirnent 1	12.4 %	0.397	0.603
Ex eriment 2	8.9 %	0.673	0.327
Ex eriment 3	14.0 %	0.226	0.774
Ex erirnent 4	12.4 %	0.408	0.591

Measured packet loss is far lower for all experiments than any other method. Across an unloaded line the packet loss was 0.8%, which indicates the effectiveness of this method. The packet loss for this experiment occurred regularly and it appears that an outside influence caused this loss.

Rapid decrease, but slow increase, in QoS resulted in low packet loss occurring in all experiments. Only the delivery of a QoS of 7200 Bps caused large loss to occur, but this can be prevented by introducing extra QoS groups. Reduction in the amount of times a high QoS is delivered can be achieved by reducing the acceptable window for an increase in QoS. For these experiments, a value of 10°/0 was used but reducing it to 5% would have resulted in far a lower delivery of a QoS of 7200 Bps. The measured loss for most experiments was between 9% and **14%.**

Jitter

Experiment Number	Average Jitter	Average Inter-Packet Arrival Time	Packet Arrival Rate
No-Traffic	1.659	179.62	5.757
Experiment	36.855	239.439	4.347
Experiment	22.441	260.38	4.114
Experiment	51.100	280.613	3.691
Exp '	38.610	60.4	4.082

Due to the high average throughput and low packet loss, the jitter value for most experiments was reasonably low. The amount of network traffic on the network can be seen on the jitter graph (figure 6.21), which reveals that fluctuations in delivered QoS correspond to peaks and troughs visible on the jitter graph. The higher the loss for a particular QoS, the higher the corresponding peak in the graph.

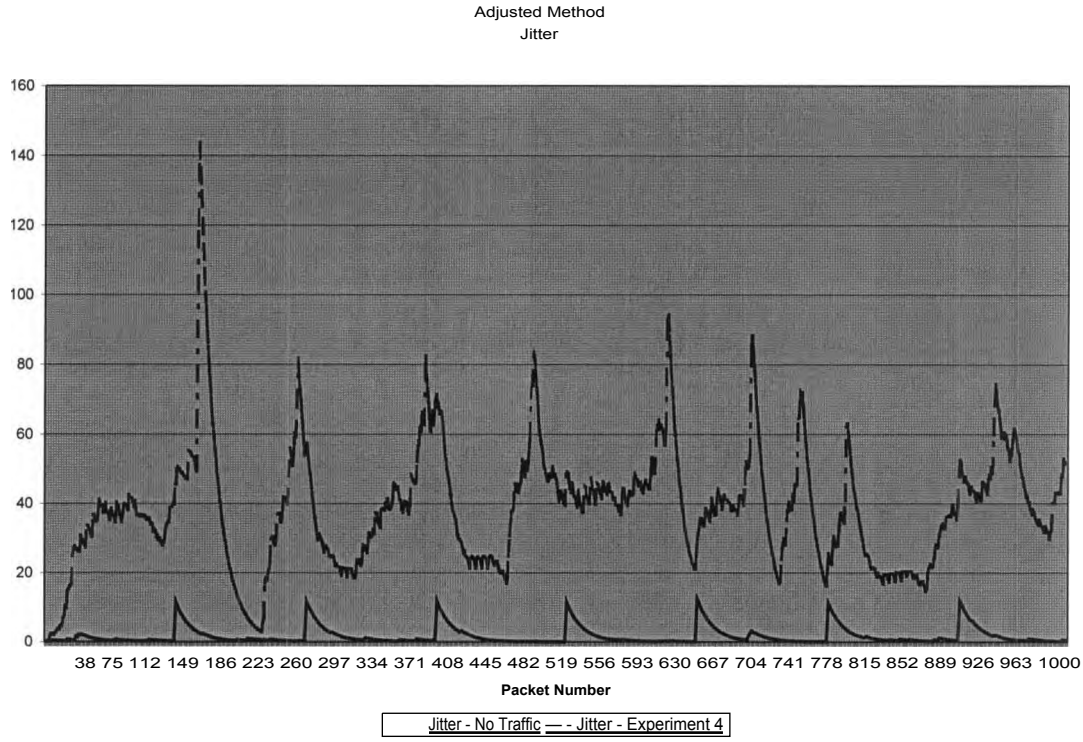

Figure 6.21. — Adjusted method: Jitter graph

Conclusion

Experiment Number	Maximum QoS Packet Count	QoS Decrease Ratio	QoS Increase Ratio
No-Traffic	0	0.250	0.750
Exce	675	0.333	0.667
Ex	0	0.369	0.631
Ex	0	0.344	0.656
Ex	113	0.349	0.651

Measured throughput is slightly lower for the *Adjusted* method than the other methods, but measured packet loss is far lower. This method was originally introduced as a means of reducing the high packet loss experienced by the packet loss and throughput methods, and reducing the number of renegotiations performed by the *Current* method.

This method is clearly the most successful of all renegotiation methods. Improvement of this method is possible by altering the threshold levels of the network-states and acceptable percentages during the lifetime of the connection. This will allow more packets to be received in time (increasing the throughput) without increasing the QoS to an unacceptable value (maintaining a low packet loss).

6.6. Concluding Remarks on Experiments and Results

The QoS renegotiation methods discussed in this chapter provide the final implementation component of the experimental system. Delivery of a constant, non-guaranteed QoS data stream is possible without any renegotiation process. The implementation of a renegotiation process within the Resource Monitor component of the Session Manager provides a mechanism allowing for the dynamic readjustment of the delivered QoS. The Resource Monitor is responsible for monitoring the received data stream and collecting various statistical data. Calculation of the measured throughput and loss over the renegotiation period is compared to the delivered QoS values, and a decision to hold, increase, or decrease the QoS is made.

Investigation of a number of different renegotiation methods based on either the measured throughput or loss value (but not both values) proved to instigate a high throughput and packet loss. Although a high delivered throughput relates to a high delivered QoS, a large packet loss causes the delivered data stream to suffer from serious degradation in quality. Increases and decreases in QoS were based on the same allowed fluctuation value, and sudden increases in the QoS corresponded to large packet loss.

The *Adjusted* Method increases the delivered QoS by a maximum of one QoS group, but decreases are permitted up to the maximum fluctuation allowed. Any renegotiation is based on both the measured throughput and loss. The measured statistics prove that this method is far superior to other methods with regard to the delivered QoS. Although the measured throughput is slightly less, the lower packet loss makes for a more impressive QoS. For this reason, the *Adjusted* Method is recommended as the primary method to be used for the renegotiation process.

Implementation of the various renegotiation methods completes the RTP Audio Application prototype. Comparison of the various renegotiation methods shows that it is possible to dynamically adjust the delivered data stream whilst maintaining high throughput and low loss values. The primary aim of the system is the delivery of a variable QoS data stream, whilst sustaining a constant information content. The RTP Audio Application discussed and implemented throughout this thesis provides a simple prototype system that allows this aim to be accomplished.

Chapter 7

Future Work and E x tensions

7.1. Introduction

The RTP Audio Application discussed in the previous chapters tests whether an adaptive flow system for the dynamic adjustment of the delivered QoS is possible. A number of extensions to the system, and also improvements to the QoS renegotiation algorithm are presented in this chapter.

The transmission of a video data stream rather than an audio stream is the initial extension proposed. This tests whether the renegotiation algorithms are generic and work for all multimedia data types or only one specific data type. Although the *Adjusted* QoS renegotiation method is the most successful methods of those investigated, a number of improvements are possible. The RTP implementation utilised for the RTP Audio Application is not complete and extensions such as multicasting and allowing a bi-directional delivery of data, while not directly improving the system would provide a larger number of networking functions.

7.2. Multimedia Extensions

A logical extension to the RTP Audio Application is to attempt to deliver video rather than audio streams. As discussed in section 2.2.1, audio and video data exhibit different loss characteristics. The design of the RTP Audio Application is intended as a component-based system providing mechanisms for the dynamic adjustment of a delivered multimedia data stream. Changing the type of data delivered should only require altering the Service Manager interface to the new media service. Transmission of video streams would test whether the *Adjusted* method is able to adjust to different multimedia data, or whether the algorithm is specifically tailored towards delivery of audio data.

Experiments involving video data require a test network with a large available bandwidth. The PPP line used for the experiments involving the RTP Audio Application is not nearly sufficient for the successful transmission of video data. Even delivery of video data at the lowest possible QoS requires an available bandwidth that is of a magnitude larger than the maximum possible transmission rate of a PPP line. Current heterogeneous IP networks, such as the Internet, do not currently possess sufficient available bandwidth to successfully deliver high quality video data streams.

The number of QoS parameters for video data is far larger than for audio data. QoS parameters such as frame size, colour-depth, frame resolution and complexity are required for each transmitted frame.

Basso *et. al.* investigate the transmission of MPEG-2 Streams over Non-Guaranteed QoS Networks. Although this work does not attempt to dynamically adjust the delivered QoS, an investigation into the delivery of RTP packets based on different schemes is presented [Busse *et. al]*.

Campbell presents an in-depth investigation of the layered model of delivering MPEG-2 video streams [Campbell *et. al* (2)]. A combination of this work and the research by Basso *et. al.* would enable delivery of a dynamically adjustable QoS of video data. Network resource availability determines whether extra enhancement layers are delivered, and fluctuations in the delivered QoS are determined according to the *Adjusted* Method. Delivery of layered MPEG-2 video streams requires the guaranteed delivery of a base-layer. Currently, provision of QoS guarantees is not possible using the RTP Library and so this would need to be added to the system.

7.3. RTP Extensions

7.3.1. Bi-directional data delivery

The RTP Audio Application is currently uni-directional, based on a traditional client/server data delivery model. The server transmits audio data and the receiver is then able to receive it. Facilitation of a peer-to-peer connection (and therefore bi-directional data transfer) would be reasonably simple to implement. Either the client or server, not both, would perform the renegotiation of the QoS. This would permit a wider range of applications, such as video-conferencing, to be constructed using dynamic QoS adjustment techniques.

7.3.2. Multicasting

Multicasting of multimedia data has been discussed in detail in section 2.1.3. An attempt to add multicasting to the RTP Library is also discussed in section 4.5. The addition of multicasting functions would require a new renegotiation method to be discovered. The *Adjusted* method works specifically in a server/client environment. Renegotiation of the delivered QoS in a multicasting environment is more complex than an unicasting environment, and more accurate results are achieved in an unicasting environment.

Mixers and translators can be used to deliver a lower QoS data stream to clients with less available bandwidth, thus allowing a specified QoS to be negotiated between the server and all mixers. The mixers are then responsible for delivering a lower QoS to all clients that require it. This method is not ideal, however, as it requires the placement of mixers at all points on the network where the client will receive a lower QoS.

7.3.3. Compression of Multimedia Streams

Most multimedia data types can be compressed by the server, transmitted across a network and then uncompressed at the receiver with very small loss in quality. Audio data can be compressed using a 4:1 ratio and larger compression of video data is also possible. For the RTP Audio Application, experiments were conducted using uncompressed audio data as the throughput, and loss values were relative to the maximum possible values. Packet size was constant for all experiments and therefore uncompressed or compressed data would result in the same calculated values.

7.4. Adjusted QoS Renegotiation Method Extensions

Although the *Adjusted* method is the most suitable renegotiation method for the delivery of audio data, it is possible to improve the method further. Increases and decreases in QoS are dependent on the current network-state and the measured throughput and loss values. There is currently no history of the packet loss and throughput measured for a particular delivered QoS over time.

For all experiments, regardless of network usage, the maximum delivered QoS experienced a high packet loss and high throughput, and the minimum QoS a low throughput and low packet loss. Consider the following situation: a delivered QoS at 7200 Bps, with a packet loss of less than 8% and an average throughput of 6950 Bps. Although the measured throughput and loss are within the acceptable window for an increase in QoS to occur, the history of delivering a QoS of 14400 Bps would indicate that regardless of network conditions, high packet loss will occur and the delivered QoS will immediately be reduced. In this situation, the acceptable window for packet loss and throughput would be reduced to a smaller amount (indicating a low network usage and a higher chance of successfully delivering the new QoS) and the delivered QoS would be maintained rather than increased.

Chapter 8

Conclusion

The delivery of a constant information content over heterogeneous IP networks is the primary aim of this thesis. This approach was taken as an alternative to the delivery of a constant QoS, which requires the type of network resource reservation that is not available over the current generation of widespread IP networks, such as the Internet. This research demonstrates that it is possible to deliver a constant information content by dynamically adjusting the delivered QoS throughout the connection lifetime. To accomplish this, research was undertaken to develop a client/server system using an adaptive-flow approach, which attempts to predict the level of network traffic, and renegotiate the delivered QoS accordingly.

A number of useful concepts for the design of future adaptive flow systems were uncovered during the design and testing process of the experimental system, and are outlined below. The hybrid design approach of the experimental system, based on a three-layered model, enables unrelated tasks to be separated, and consequently reduces the complexity involved in implementing applications of this scale. To permit the maximum degree of flexibility throughout the system, all threshold values were designed to be configurable. This proved to be a strong point in the design of the test environment, and facilitated tests that in many cases were unforeseen during the design phase.

RTP and RTCP were chosen as the transport protocols due to their timestamping, sequence numbering, and payload type identification functions. RTP has the added advantage of being deliverable across the current generation of IP networks (version 4), without requiring large-scale modifications to the lower-level devices, such as routers and switches. Current network protocols need not be extended as RTP makes use of underlying delivery mechanisms such as UDP. The use of RTP and RTCP as networking protocols facilitates a generic transport mechanism upon which a higher-level adaptive-flow system can be developed.

The timestamping function of RTP provides an accurate mechanism for monitoring the throughput and jitter of the delivered data stream, as well as for allowing for a reconstruction of the original timing properties of the recorded data. Sequence numbers are required to determine the packet loss of the data stream, and also allow for the original packet order to be reconstructed at playout time. Due to the packet-switched nature of IP networks, data packets may not arrive at the client in the same order that they are sent by the server

Implementation issues relating to synchronisation and buffering occurred not only at the intra-system boundary between the application and media device, but also at the inter-system interface between the data server and the client. Although the behaviour of the data flow is reasonably predictable, the level of competing network traffic is not. A constant QoS is delivered for a pre-arranged renegotiation period. It is therefore possible to predict the required network resources for the following renegotiation period, although changes in the level of competing network traffic might reduce the quality of the prediction. The smoothing nature of the fluctuation value, introduced during the investigation of the renegotiation algorithms, improved the predictability of the renegotiation process.

This research makes underlying assumptions both about the nature of contending network traffic in the test environment, and about the nature of the multimedia data transmitted. As discussed in the Quality of Service issues section, different multimedia data types display different loss characteristics, and the QoS renegotiation methods may not be able to manipulate a video flow as successfully as an audio flow. The renegotiation method put forward as being highly effective in the test environment, might not be as effective in a situation where the real network differed from these assumed characteristics. Consequently, our approach to developing the algorithm, rather than the actual algorithm, should be adopted with the greater measure of confidence. The algorithm should be adapted for real network conditions. Fine-tuning of the renegotiation methods, and the thresholds determining acceptable packet loss and throughput, will certainly improve measured results in a real network.

Some of the most significant contributions of this thesis are in the area of the renegotiation algorithms themselves. Different renegotiation methods exhibit different characteristics both in terms of the level of delivered QoS, and in terms of measured throughput and loss. Methods based only on either one of loss or throughput are not able to accurately renegotiate QoS. One needs to base future adjustments on both values. The *Adjusted* method, devised during this research, is an accumulation of the advantages of all methods examined, whilst attempting to minimise the disadvantages. Quantitatively, we have shown that the *Adjusted* method does provide the best trade-off between throughput and loss. The packet loss value is an indication of the reliability of the delivered QoS; the lower the packet loss, the higher the chance that all delivered packets will arrive, and that an uninterrupted data stream will be enjoyed by the user. The results measured from the *Adjusted* method demonstrate that the primary aim of the research, the delivery of a variable QoS data flow from a constant information content, can be achieved. It is also significant to note that the streaming application was constructed using currently available, widely used, I? networking structures.

The overall assessment of the experimental system attempted not only to determine the effectiveness of each QoS renegotiation method, but also to provide an indication of the viability of the overall adaptive flow approach. A primary objective of constructing the experimental system was met when an audio

stream could be delivered to a client while the intermediate network load and bandwidth was altered, with a consequent adjustment in quality, but not in content, of the output.

The adaptive flow approach to the delivery of multimedia content, advocated throughout this thesis, is aimed at delivering a constant information content to users, regardless of the bandwidth they are able to afford, or the circumstances that might mitigate against a reliable, dedicated network between them and the content server. While serving low QoS content to less privileged clients, the approach provides for the same content to be delivered at a higher QoS to users with larger, more expensive, and more reliable network connections. The current Internet, and the kinds of networks that are likely to dominate in developing countries for some time to come, are ideal application areas.

Appendix 1.a.

RTCP Socket Code — Header File

```
/* RTCP Library

RTCP Socket Code - Header file

Developed to be used in conjuction with the RTP library developed by the
CNR IASI Netlab

Paul Littlejohn, 17 March 1998 */

#ifndef  RTCP LIB HH
#define RTCP LIB HH

#include <errno.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <iostream.h>
*include <arpa/inet.h>


/*************************************************************************
                                RTCP SOCKET


class RTCP_Socket
{
  protected:
    struct sockaddr in local addr; // the local IP address and port
                                   // number
    int fd sock;   // socket for the client to connect to
    int rec sock; // socket to receive data on
```

```
  public:
    RTCPSocket();
     RTCP Socket();
    int Bind();          // Binds the socket to local addr
1;

/*****************************************************************************
                            RTCP RECEIVE SOCKET


class RTCP Receive Socket : public virtual RTCP Socket
{
  private:
  public:
    RTCP Receive Socket();
    -RTCP Receive Socket();
    int Receive(void *, int); // Receive data on the given socket
    int Accept();             // Accept connections from clients
1;



                            RTCP SEND SOCKET


class RTCP_Send_Socket : public virtual RTCP_Socket

  private:
  protected:
    struct sockaddr in remote addr; // IP address and port number to send
                                    // data to and make TCP/IP connection
  public:
    RTCP Send Socket::RTCP Send Socket();
    RTCP Send Socket::-RTCPSendSocket();
    int Send(void *, int);    // Send the control information
    int Connect(char *, unsigned short int);
            // Create a TCP/IP connection with the remote addr
1;
```

```
/*****************************************************************************
                          RTCP SEND RECEIVE SOCKET


class RTCP Send Receive Socket : public RTCP Send Socket,
                                 public RTCP Receive Socket

  private:
  public:
    RTCP Send Receive Socket(unsigned short int);
    -RTCP Send Receive Socket();
1;

#endif
```

Appendix 1.b.

RTCP Socket Code — Class Libraries

```
/* RTCP Library

RTCP Socket Code - Class Libraries

Developed to be used in conjuction with the RTP library developed by the
CNR IASI Netlab

Paul Littlejohn, 17 March 1998 */


#include <errno.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <iostream.h>
#include <arpa/inet.h>

#include "rtcpsocket.hh"

  ***************************************************************************
                                RTCP SOCKET


RTCPSocket::RTCP_Socket()
  {
    // create a TCP/IP socket
    fd sock = socket(AF INET, SOCK STREAM, 0);
    if (fdsock -- -1)
      cerr << "RTCP Socket() failed to create correctly\n";
```

```
RTCP Socket::-RTCP Socket()

    // close the socket and check for errors
    int closeval = close(fd sock);
    if (closeval == -1)
      cerr << "-RTCP Socket fdsock did not close correctly\n";


int RTCP Socket::Bind()

    // bind the socket to the local IP address and port number
    int bindval = bind(fd sock, (struct sockaddr *) &local addr,
                       sizeof(struct sockaddr in));
    if (bindval == -1)
      cerr << "RTCPSocket::Bind() failed with errno   << errno << \n";
    return bindval;



/*****************************************************************************
                            RTCP RECEIVE SOCKET



RTCPReceiveSocket::RTCPReceiveSocket()

  1

RTCP Receive Socket::-RTCP Receive Socket()

  1

int RTCP Receive Socket::Receive(void *buffer, int length)

    fd set   rfds;
    struct timeval t;
    t.tvsec = 0;
    t.tvusec = 0;
```

```
    FD ZER0(&rfds);           // clear the receive socket set
    FD SET(rec sock, &rfds); // include rec sock in the receive socket
                             // set

    select(rec sock+1,&rfds,NULL,NULL,&t);
            // check the state of the receive socket set

    if(FD ISSET(rec sock, &rfds)) // if rec sock is set then there is
                                  // data available at the socket

        int recvval;
        recvval = recv(rec sock, buffer, 128, 0);
            // receive up to a maximum of 128 bytes of data (control
            // information is only 8 bytes)
        if (recvval == -1)
          cerr << "RTCPReceiveSocket::Receive() failed\n";
        return recvval;
      1
    else
      {
        return 0;



int RTCP Receive Socket::Accept()

    RTCP Socket::Bind();
      // Firstly bind the socket to the local IP address and port

    int listenval = listen(fd sock, 10);
      // listen for any connections

    struct sockaddr in temp;
    int size = sizeof(struct sockaddr);

    rec sock = accept(fd sock, (struct sockaddr*) &temp, (unsigned int *)
                      &size);
      // create a socket to receive data (rec sock)
```

```
    if (rec sock == -1)
      cerr << "RTCP Receive Socket::Accept() failed\n";
    return rec sock;



 ********************************************************************************
                              RTCP SEND SOCKET
********************************************************************************

RTCPSendSocket::RTCPSendSocket()
  {
  }

RTCP Send Socket::-RTCP Send Socket()
  {
  }

int RTCP Send Socket::Send(void *buffer, int length)

    struct sockaddr in temp;
    unsigned int tempint = sizeof(temp);

    int sendval = send(fd sock, buffer, length, 0);
      // send a buffer of data using the given socket

    if (sendval == -1)
      cerr << "RTCP Send Socket::Send() failed\n";
    return sendval;
  1

int RTCP Send Socket::Connect(char* host, unsigned short int port)
  {
    hostent *h;
    h = gethostbyname(host);
      // get the hostname of the remote host to connect to

    memcpy((void *)&remote addr.sin addr.s addr, (void *)h->h addr,
            h->hlength);
      // copy the host address into the remote addr structure
```

```
    remote addr.sin family = AF INET;
    remote addr.sin port = ntohs(port);
      // update all fields of the remote addr structure
    int connectval;

    connectval = connect(fd sock, (struct sockaddr*) &remote addr,
                         sizeof(sockaddr in));
      // connect to the remote address

    if (connectval == -1)
      // check if the connection was successful, if not print an error
      // message

        cerr << "RTCP Send Socket::Connect() to
             << inet ntoa(remote addr.sin addr) << " on port "
             << ntohs(remote addr.sin port) << " failed with errno
             << errno << "\n";

    return connectval;


/*****************************************************************************
                          RTCP SEND RECEIVE SOCKET


RTCPSendReceiveSocket::RTCPSendReceiveSocket(unsigned short int
local port)

    hostent *pHostent;
    char local_host[64];
    int retval = gethostname(local host, 64);
    if (retval == -1)
      cerr << "gethostname() failed\n";
    pHostent = gethostbyname(local host);
      // at instantiation time we need the local host name to set up the
      // socket correctly
```

```
    memcpy((void *)&local addr.sin addr.s addr, (void *)pHostent->haddr,
           pHostent->h length);
    local addr.sin port = htons(localport);
    local addr.sin family = AF INET;
      // copy the local host information into the local addr structure to
      // allow the networking functions to operate correctly


RTCPSendReceiveSocket::-RTCPSendReceive Socket()
  {
  }
```

Appendix 2.a.

QoS Mapper — He a der File

```
/* gosmapper.hh

  These QoS renegotiation methods are to be used in conjuction with the
  RTP Library developed by the CNR IASI Netlab.

  This class is responsible for maintaining the available QoS values for
  a particular application. These QoS values are then used for the
  dynamic renegotiation of the delivered QoS

  Paul Littlejohn , Rhodes University , 20 July 1998 */


#ifndef _QOS_MAPPER_HH_
#define  QOS MAPPER_HH_

#include "types.hh"
#include "statistic.hh"

%define MAX DATA_ITEMS        50
#define MAX_QOS ITEMS         32
#define MIN SAMPLING RATE     900
#define MAX SAMPLING RATE     3600

typedef struct
  {
    unsigned int sent timestamp;
    unsigned int received timestamp;
    unsigned int packet_size;
    unsigned int sequence number;
    packet info;
      // used to store the relevant information for each received data
      // packet

class QoS Mapper

private:
  gos_element*    (los table[MAX QOS ITEMS];
  packet info*    data table[MAX DATA ITEMS+1];

  FILE*           various;

  Statistic*      pstat_throughput;
  Statistic*      pstat loss;
  Statistic*      pstat jitter;
  Statistic*      pstat qos values;
  Statistic*      pstat packet rate;

  int       total_bytes_received;
  int       total_packets_received;
  int       total_packet_loss;
```

```
  // current values of max, min and average throughput
  double    max_throughput;
  double    min_ throughput;
  double    ave_throughput;

  // current values of max, min and average throughput
  double    max_loss;
  double    min_loss;
  double    ave_loss;

  unsigned int    max jitter;
  unsigned int    min jitter;

  // packet loss values
  int       never_arrive_packets;
  int       out_of_order packets;
  int       late packets;

  // Number of packets received before the maximum QoS reached
  int       max qos packet count;

  int       increase in gos;
  int       decrease ingos;

  // current position in the QoS table
  int       current qos;

  // qos of the packet that has just been delivered
  int       delivered qos;

  // current position in the data table
  int             current data item;

  // current measured throughput value
  double    current throughput;

  // current measured loss value
  double    current loss;

  // current measured jitter value
  unsigned int    current jitter;

  // initial timestamp used for measuring average values
  unsigned int    initial timestamp;

  // Updates current_gos value according to specified algorithm
  int       Adjust QoS();

  int       Packet Delay();
  /* Checks for the three different types of packet loss :
1. Packets that never arrive
2. Packets that are too late to be played
3. Packets that arrive out of order and the preceeding packet has been
played

2 & 3 increase total throughput and packets received, they use bandwidth
and are therefore included in the statistics.

Packet Delay is called for every packet. Only packets that meet the delay
criteria are entered into the data table. Therefore the throughput
```

```
methods already have checked for packet loss. Also measures the delay for
each received packet. */


  // Calculate the max, min and ave loss and throughput
  int             Calc Loss and Throughput Stats();


  /7 Different algorithms that have been used to adjust the qos

  int       Change Every 20 Seconds();
  /* Every twenty seconds the current qos is changed. Used as a
demonstration technique */

  int             Total_Throughput();
  /* Total_ Throughput measures the bandwith used of received packets and
adjusts the qos according to available resources. If the QoS is set too
high, simply adjust the qos to a level that can be accomodated. An
incremental increase in qos will be attempted if the qos is too low */

  int       Total Packet Loss();
  /* Similar to total_throughput, except that instead of measuring
bandwidth, packet loss is measured. If packet loss occurs then the qos is
decreased until no packet loss is encountered. If no packet loss then qos
increased until packet loss occurs.
Three types of packet loss
1. Packets that never arrive
2. Packets that arrive too late to be played,
3. Packets that arrive out of order and the preceeding packet has already
been played */

  int       Current Throughput();
  /* The current throughput is measured and the QoS is adjusted according
to this value. If the current Throughput is greater than or equal to the
current QoS value then it is adjusted upwards. If the throughput is lower
than the current (expected) throughput then the QoS is adjusted
accordingly. This is similar to a smoothed throughput but with a
smoothing variable equal to one packet, regardless of the length of the
timing interval */


  int       Smoothed Packet_Loss();
  /* Smoothed Packet_Loss attempts to determine the level of QoS based on
the loss of packets throughout the lifetime of the connection. A smoothed
loss variable determines the level of fluctuation allowed in the
adjustment of the new QoS. Another variable can also be introduced to
allow the smoothed loss to only be monitored over a certain time period,
not the entire lifetime. */

  int       Smoothed Throughput();
  /* Similar to smoothed_packet_loss, except instead of measuring packet
loss, total bandwidth usage is measured */

  int       Congested Throughput();
  /* Defines three variables , congested, loaded and unloaded. If
throughput is currently in the acceptable congested window then qos is
maintained, otherwise reduced to a value within the window. This can be
used as a user specified minimum qos value. User states that this is the
minimum QoS that I am willing to receive. Would rather lose packets than
drop below this level. */
```

```
  int       Congested Packet Loss();
  /* Same as congested throughput except works on packet loss. Congested
and uncongested defined as a percentage of total packets delivered. */

  int       Adjusted Method();
  /* Bases the renegotiation process on both the packet loss and
throughput values. Attempts to minimise packet loss and maximise
throughput by increasing the QoS by only a single QoS group, but
reductions are allowed up to the maximum fluctuation determined by the
current network state. */


public:
  QoS Mapper();
  -QoS Mapper();

  // Outputs the entire gos_table and the current qos value
  void PrintQoSTable();

  // Outputs the entire packet_table (packet sizes and timestamps)
  void Print Data Table();

  // Receives packet size from rtpd sap, updates the packet_table and
then
  // adjusts the current gos value
  int Add QoS Data(unsigned int,    // packet size
                   unsigned int,    // sent timestamp
                   unsigned int);   // sequence number

  // Called by client application to receive new QoS values
  qos element GetCurrentQoS();

  // Called by data receiver to change the delivered_qos variable so that
  // measurements are able to be taken
  int Set Current QoS(qos element*);

};

#endif
```

Appendix 2.b.

QoS Mapper — Cla s s Libraries

```
/* gosmapper.cc

  These QoS renegotiation methods are to be used in conjuction with the RTP
  Library developed by the CNR IASI Netlab.

  This class is responsible for maintaining the available QoS values for a
  particular application. These QoS values are then used for the dynamic
  renegotiation of the delivered QoS

  Paul Littlejohn , Rhodes University , 20 July 1998 */

#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <math.h>
#include <unistd.h>
#include <stdio.h>

#include "sys_time.hh"
#include "qos_mapper.hh"
#include "types.hh"

#define RTP_HEADER 40
#define IP_HEADER  20
#define CONTROL     8

extern Statistic Manager The_Statistic_Manager;


QoS Mapper:: QoS_Mapper()
{
  for (int i = 0; i <= MAX_DATA_ITEMS; i++)
    {
      data table[i] = NULL;


  for (int i = 0; i < MAX_QOS_ITEMS; i++)

      qostable[i] = NULL;


  // in order for all the qos elements to be in order of preference (lowest
  // to highest) a few checks are needed
  int index = 0;
  for (int rate = MIN SAMPLING RATE; _rate <= MAX SAMPLING RATE; rate *= 2)

    for (int _samplesize = 1; samplesize <= 2; _samplesize ++)
      {
      for (int _channels = 1; channels <= 2; _channels ++)

          if ((_channels == 1) && (_samplesize == 1) && (_rate >
              MINSAMPLINGRATE))

              gos_table[index-1] = new gos_element;
              gos_table[index-1]->rate = _rate;
              gostable[index-1]->samplesize = _samplesize;
```

```
            clos_table[index-1]->channels = channels;
            gostable[index-1]->bandwidth = rate * samplesize * channels;

        else if ((_channels == 2) && (_samplesize == 2) && (_rate <
                 MAXSAMPLINGRATE))

            clos_table[index+1] = new gos_element;
            clos_table[index+1]->rate = rate;
            gos_table[index+1]->samplesize = _samplesize;
            gos_table[index+1]->channels = _channels;
            gostable[index+1]->bandwidth = _rate * samplesize * channels;

        else

            qos table[index]   new qos_element;
            clos_table[index]->rate = rate;
            gos_table[index]->samplesize = _samplesize;
            qos table[index]->channels = channels;
            gostable[index]->bandwidth = _rate * samplesize * channels;
          1
        index ++;
      }



total_packet_loss = 0;
total_packets_received = 0;
total_bytes_received = 0;

current_data_item = -1;
current_gos = 0;
current_throughput = 0.0;
current loss = 0.0;

max_loss = 0.0;
min loss = 0.0;
ave_loss = 0.0;

max_throughput = 0.0;
min_throughput = 30000.0;
ave throughput = 0.0;

max_jitter = 0;
min jitter = 30000;

never_arrive_packets = 0;
out_of_order_packets = 0;
late packets = 0;

maxgospacketcount = 0;

increase_in_gos = 0;
decreaseingos = 0;

char name[128];
char mname[64];
gethostname(mname,64);

strcpy(name,"Probe.");
strcat(name,mname);
strcat(name,".QoS_Mapper_Throughput");
pstat throughput = TheStatisticManager.QueryNewStat(name);

strcpy(name,"Probe.");
strcat(name,mname);
strcat(name,".QoSMapperLoss");
pstat loss = TheStatisticManager.QueryNewStat(name);
```

```
  strcpy(name,"Probe.");
  strcat(name,mname);
  strcat(name,".QoS_Mapper_Jitter");
  pstat jitter = TheStatisticManager.QueryNewStat(name);

  strcpy(name,"Probe.");
  strcat(name,mname);
  strcat(name,".QoSMapperQoS_Values");
  pstat qos values = TheStatisticManager.QueryNewStat(name);

  strcpy(name,"Probe.");
  strcat(name,mname);
  strcat(name,".QoS_Mapper_PacketRate");
  pstat packet rate = TheStatisticManager.QueryNewStat(name);

  strcpy(name,"Probe.");
  strcat(name,mname);
  strcat(name,".QoSMapperVarious");
  various = fopen(name, "w");

  PrintQoSTable();


QoS24Apper::-QoSidApper()
{
  PrintDataTable();
  int index = 0;
  while (clos_table[index] != NULL)
    {
      delete gos_table[index];
      index++;


  for (index = 0; index <= MAX DATA ITEMS; index++)
    {
      if (data_table[index] != NULL)
        delete data_table[index];
      index++;


  double total_time = (data_table[current_data_item - 1]->received_timestamp -
                       initial_timestamp) / 1000.0;
  ave_throughput = (double)total bytes received / total_time;
  ave_loss = (double)total_packet_loss / (double)total_packets_received;
  double ave_jitter = (double)total_packets_received / total_time;

  fprintf(various, "Total Bytes Received\t: %d\n", total_bytes_received);
  fprintf(various, "Total Time\t\t: %f\n", total_time);
  fprintf(various, "Total Packets Received\t: %d\n\n", total_packets_received);

  fprintf(various, "Max Throughput\t: %f\n", max_throughput);
  fprintf(various, "Min Throughput\t: %f\n", min_ throughput);
  fprintf(various, "Ave Throughput\t: %f\n\n", ave_throughput);

  fprintf(various, "Total Packet Loss\t: %d\n", total_packet_loss);
  fprintf(various, "Average Packet Loss\t: %f\n\n", ave_loss);


  if (total_packet_loss > 0)
    {
      fprintf(various, "Breakdown of packet loss\n");
      fprintf(various, "Lost Packet Ratio\t: %f\n", (double)never arrive packets
              / (double)total_packet_loss);
      fprintf(various, "Out of Order Packet Ratio\t: %f\n",
             (double)out_of_order_packets / (double)total_packet_loss);
      fprintf(various, "Late Packet Ratio\t: %f\n", (double)late_packets /
             (double)total_packet_loss);
```

```
  fprintf(various, "Increase in QoS Total\t: %d\n", increasein_qos);
  fprintf(various, "Increase in QoS Ratio\t: %f\n", (double)increase_in_qos /
         ((double)increase_in_qos + (double)decrease_in_qos));
  fprintf(various, "Decrease in QoS Total\t: %d\n", decrease_in_qos);
  fprintf(various, "Decrease in QoS Ratio\t: %f\n\n", (double)decrease_in_qos /
         ((double) increase in qos + (double)decreaseinqos));

  fprintf(various, "Number of packets before max QoS reached\t: %d\n\n",
         maxqospacketcount);

  fprintf(various, "Maximum Jitter\t: %d\n", max_jitter);
  fprintf(various, "Minimum Jitter\t: %d\n", min_jitter);
  fprintf(various, "Average Jitter\t: %f\n\n", ave_jitter);

  fclose(various);

  pstat_throughput->WriteToFile(NULL);
  pstat_loss->WriteToFile(NULL);
  pstat_jitter->WriteToFile(NULL);
  pstatqos_values->WriteToFile(NULL);
  pstatpacketrate->WriteToFile(NULL);

  delete pstat_throughput;
  delete pstat_loss;
  delete pstat_jitter;
  delete pstat_qos_values;
  delete pstat packet rate;


void QoS Mepper::Print_QoS_Table()
{
  int index = 0;
  cout << "QoS Mapping table for audio application\n\n";
  cout << "Rate\tSamplesize\tChannels\tBandwith\n";
  while (qostable[index] != NULL)

      cout << gos_table[index]->rate << "\t\t"
           « qos_table[index]->samplesize << "\t\t"
           << qos_table[index]->channels << "\t"
           << qos_table[index]->bandwidth << "\n";
      index++;
    1
  cout << "\nCurrent QoS value:\n";
  cout << qos_table[current_qos]->rate << "\t\t"
       << gos_table[current_gos]->samplesize << "\t\t"
       << qos_table[current_qos]->channels << "\t"
       << gostable[currentgos]->bandwidth << "\n";


void QoS MApper::Print_pata_Table()

  int index = 0;
  cout << "Data table for audio application\n\n";
  for (index = 0; index <= MAX DATA ITEMS; index++)

      if (data_table[index] != NULL)
        cout << data_table[index]->senttimestamp << "\t\t"
             << data_table[index]->receivedtimestamp << "\t"
             << data_table[index]->packet_size << "\t"
             « datatable[index]->sequencenumber << "\n";



int QoS Mapper::Add_QoS_pata(unsigned int pkt_size, unsigned int sent_timestamp,
unsigned int seg_num)

  // check if this is the first packet ever received
```

```
int first = 0;
if (data_table[1] == NULL)
  first = 1;

// if we are going to store more than MAX DATA ITEMS we need to have a
// circular buffer to prevent overflow
if (current_data_item == MAXDATA_TTEMS)
  current_data_item = 1;
else

    current_data_item++;
    datatable[currentdataitem] = new packet info;


// Calculate current timestamp
sys_time temp_time = GetActualSystemTime();
unsigned long long int current_temp = (unsigned long long int) temp_time;
unsigned int m_current = 0;
unsigned int n_current = 0;
unsigned int count_current = 0;
current_temp = current_temp / 1000;
for (unsigned int i_current = 1; i_current <= 1000000000; i_current *= 10)

    m_current = current_temp % 10;
    current_temp = current_temp / 10;
    n_current = m_current * i_current;
    count_current = count_current + n_current;
  1
unsigned int timestamp = count_current;

if (first == 1)
  {
    cout << "******** First packet *********\n";
    initial timestamp = timestamp;


// packet size does not include overhead etc on the packet, we are only
// interested in the rtp data size, includes rtp_header
// pkt_size -= 8184;

// Have to hard code this value into the system in order to correctly
// measure the throughput
pkt size = 1024;

// the first packet is received at current time of 0, this packet did not
// take zero seconds to get here therefore we must discount it
if (current_data_item != 0)
  total_bytes_received = total_bytes_received + pkt_size;

cout << "Current Data Item\t: " << current_data_item << "\n";

data_table[current_data_item]->receivedtimestamp = timestamp;
data_table[current_data_item]->packet_size = pkt_size;
data_table[current_data_item]->sent_timestamp = sent_timestamp;
data_table[current_data_item]->seguencenumber = seq num;

if (current_data_item == MAX DATA ITEMS)
  {
    cout << "Making a copy into Position 0\n";
    data_table[0]->received_timestamp = timestamp;
    data_table[0]->packetsize = pkt_size;
    data_table[0]->sent_timestamp = sent_timestamp;
    data_table[0]->seguencenumber = segnum;


return 0;
1
```

```
qos_element QoS MApper::Get_Current_QoS()

  qos_element* element;
  element = new qos element;

  while (qostable[currentcios]    NULL)
    {
      if (current qos    (MAX QOS ITEMS - 1))
        current_qos = 0;
      else
        currentqos++;


  memcpy(element, gos_table[current_cios], sizeof(qos_element));
  return *element;
1


int QoS MApper::Set_Current_QoS(clos_element* element)

  long rate = element->rate;
  short samplesize = element->samplesize;
  short channels = element->channels;

  int index = 0;
  while (qos_table[index] != NULL)
    {
      if ((qos_table[index]¯>rate == rate) &&
          (qos_table[index]->samplesize == samplesize) &&
          (qos_table[index]->channels == channels))
        break;
      else
        index++;
    1
  delivered qos = index;

  Adjust_QoS();
  return 0;



int QoS Mepper::Packet_pelay()

  totalpacketsreceived++;

  // if we haven;t received any packets yet
  if ((current_data_item == -1) II (current_dataitem == 0))
    return 0;

  // Check for :
  // 1.   Packets that never arrive

  int diff;
  if (current_data_item == 0)
    diff = datatable[current_data_item]->sequence_number
           data_table[MAX_DATAITEMS]->sequence_number;
  else
    diff = data_table[current_data_item]->sequence_number -
           datatable[current_data_item - 1]->sequencenumber;

  if (diff > 1)

      neverarrive_packets += diff;
      total packet loss += diff;


  // 2.   Packets that arrive out of order and the preceeding packet has played
```

```
  if (diff < 0)

      out_of_order_packets += abs(diff);
      total packet loss += abs(diff);

      delete data_table[current_data_iteml;
      current_data_item--;
      return 0;


  // 3. Packets that are too late to be played

  // amount of time after packet was supposed to arrive that it is
  // acceptable for it to be played out, defined as milliseconds
  double acceptable_percentage = 0.5;

  // inter packet delay time on sender side;
  int expected_time = 0;
  expected_time = data_table[current_data_item]->sent_timestamp -
                  data_table[current_data_item - 1]->senttimestamp;

  // inter packet arrival time on receiver side
  int actual_time = 0;
  actual_time = data_table[current_data_item]->receivedtimestamp -
                data_table[current_data_item - 1]->receivedtimestamp;

  pstat_jitter->Add((double) actual_time);
  current jitter = actual_time;

  // diff is used to determine number of packets that never arrive and
  // therefore the total time used

  if (actual_time > (expected_time + (expected_time * acceptable_percentage)))

      late_packets ++;
      total packet loss++;
      cout << "REDUCING CURRENT DATA ITEM LATE PACKET\n";
      return 0;


  cout << "Packet Delay total packet loss : " << total_packet_loss << "\n";
  return 1;
1

int QoS MApper::Adjust_QoS0
{
  // Choose the renegotiation method to be used

  // ChangeEvery20_Seconds();

  // Total_Throughput();
  // Total_Packet_Loss();

  // Current Throughput();

  // Smoothed_ Throughput();
  // Smoothed_Packet_Loss();

  // Congested_ Throughput();
  // Congested_Packet_Loss();

  AdjustedMethod();

  Calc_Loss_and_Throughput_Stats();
  return 0;


int QoS HApper::Calc_Loss_and_Throughput_Stats()
```

```
  static int not_found = 1;

  if (current_ throughput > max_ throughput)
    max throughput = current throughput;

  if (current_ throughput < min_ throughput)
    min throughput = current throughput;

  if (current loss > max loss)
    max loss = current loss;

  if (current_loss < min_loss)
    min loss = current_loss;

  if ((qos_table[current_gos + 1] == NULL) && (not_found))

      max_gos_packet_count = total packets received;
      not_found = 0;


  if (current_jitter > max jitter)
    max jitter = current_jitter;

  if (current_jitter < min jitter)
    min jitter = current_jitter;

  return 0;



/*************** CHANGE EVERY x SECS ********************/

int QoS MApper::Change_Every_20_Seconds()
{
  if (data_table[0] == NULL)
    return 0;

  static int current_difference = 20;

  unsigned int time = data_table[current_data_item]->received_timestamp -
                      data_table[0]->receivedtimestamp;
  time 1= 1000;
  cout << "Elapsed time\t: " << time << "\n";

  if (time > current difference)

      cout << "CHANGING QOS\n";
      current_gos++;
      if (gos_table[current_gos] == NULL)
        current qos = 0;
      current_difference += 20;


  return 0;
1


/******** ******* ****** TOTAL********** ********** ****/

int QoS HApper::Total_Throughput()
{
  if (data_table[0] == NULL)
    return 0;


  double acceptable_percentage = 0.10;
  // if the throughput is within an acceptable_percentage of the current QoS
```

```
// (stored as current qos) then QoS is increased

// The last data_item that the QoS was smoothed at
static int last data item = 0;

// The number of bytes received in the current interval
static int interval bytes received = 0;

unsigned int previous qos = current qos;
unsigned int current bw = 0;
unsigned int current pos = 0;

unsigned int time;

interval_bytes_received += data_table[current_data_item]->packet_size +
                           RTP HEADER + TPHEADER + CONTROL;

time = data_table[currentdataitem]->receivedtimestamp - initialtimestamp;

if (time > 0)

  double total_time = time / 1000.0;
  double throughput = interval_bytes_received / total_time;
  pstat_throughput->Add(throughput);
  current_throughput = throughput;
  cout << "Throughput\t: " << throughput << " kbps\n";
  pstatqosvalues->Add((double) qostable[currentqos]->bandwidth);

  if (throughput > (qos_table[delivered_qos]->bandwidth -
                   (qos_table[delivered_qos]->bandwidth *
                    acceptable_percentage)))
    { // increase QoS by one group, throughput within acceptable amount
        cout << "******** Increasing QoS ********\n";
        increase in qos++;

        // convert prey qos value into a position
        double previous_pos = (log10(qos_table[previous_qos]->bandwidth /
                               qostable[0]->bandwidth) / log10(2));

        // increase by one QoS group
        if (previous_pos < 4.0)
          current_pos = int(previous_pos + 1.0);
        else
          current pos = int(previous_pos);

        currentbw = int(qostable[0]->bandwidth * int(pow(2, current pos)));

        int index = 0;
        while (qos_table[index]->bandwidth != current_bw)

            index++;
          }
        current qos = index;
    1
  else
    { // must reduce the QoS to the closest QoS to the measured throughput
        cout << "********** Reducing QoS ***********\n";
        decrease in qos++;

        int index = 0;
        while ((qos_table[index] != NULL) && (qos_table[index]->bandwidth <
               throughput))

            index++;
          1
        if (index > 0)
          index--; // reduce by one or else we will increase the QoS
```

```
        current qos = index;


    // if the throughput was greater than max then we have gone OB's , need to
    // reduce by one
    if (qos_table[current_qos] == NULL)
      while (qostable[currentqos] == NULL)
        { current qos -= 1; }

    // Now find the most desired QoS at the current throughput
    while ((qostable[currentqos + 1] != NULL) &&
           (gos_table[current_qos]->bandwidth ==
            qos_table[current_qos + 1]->bandwidth))
      { current qos++; }

  return 0;


int QoS_MApper::Total_Packet_Loss()

  if (data_table[0] == NULL)
    return 0;

  double minimum_loss = 0.02;
  double maximum_loss = 0.10;
  double loss = 0.0;

  unsigned int previous qos = current qos;
  unsigned int current_pos;
  unsigned int current_bw = 0;

  // The number of bytes received in the current interval
  static int interval bytes received = 0;

  unsigned int time;

  // to calculate the throughput accurately we must discard all packets
  // after renegotiation has occurred until the server begins delivering
  // packets at the requested rate
  if (delivered qos != current qos)

      cout << "****** Waiting for QoS to change      *\n";
      return 0;


  // Calculate the percentage of lost packets
  loss = (double)total_packet_loss / (double)total_packets_received;
  interval_bytes_received += data_table[current_data_item]->packet_size +
                             RTP HEADER + ID HEADER + CONTROL;

  time = datatable[currentdataitem]->receivedtimestamp - initial timestamp;

  if (time > 0)

      double total_time = time / 1000.0;
      double throughput = interval_bytes_received / total_time;
      pstat_throughput->Add(throughput);
      current throughput = throughput;


  pstat_loss->Add(loss);
  cout << "Packet Loss\t: " << loss <<"\n";
  pstatqosvalues->Add((double) clostable[currentqos]->bandwidth);

  // Check if we need to increase the QoS
  if (loss < minimum_loss)
  {
```

```
    cout << "********** Increasing Qos **********\n";
    increase in qos++;

    // convert prey qos value into a position
    double previous_pos = (log10(qos_table[previous_qos]->bandwidth
                          clos  table[0]->bandwidth) / log10(2));

    // check if already at minimum QoS, reduce by one QoS group
    if (previous_pos < 4)
      current pos     int(previous_pos + 1.0);
    else
      current pos = 4;

    current low = int(clostable[0]->bandwidth * int(pow(2, current pos)));

    int index = 0;
    while (clos_table[index]->bandwidth != current_bw)

        index++;

    current qos = index;


  // Check if we need to decrease the QoS
  if (loss > maximum_loss)
    1
      cout << "********** Reducing QoS **********\n";
      decrease in qos++;

      // convert prey qos value into a position
      double previous_pos = (log10(qos_table[previous_qos]->bandwidth /
                           clos  table[0]->bandwidth) / log10(2));

      // decrease by one QoS group
      if (previous_pos > 0)
        current_pos = int(previous_pos - 1.0);
      else
        current pos = 0;

      current_bw = int(clostable[0]->bandwidth * int(pow(2, current pos)));

      int index = 0;
      while (clos_table[index]->bandwidth != current_bw)
        {
          index++;

      current qos = index;


  // make sure we don;t go ob's , bring it down if too much
  if (gos_table[current_clos] == NULL)
    while (clos_table[current_clos] == NULL)
      { current qos -= 1; }

  // Now find the most desired QoS at the current throughput
  while ((qostable[currentclos + 1] != NULL) &&
          (clos  table[current_clos]->bandwidth ==
           clos  table[currentclos + 1]->bandwidth))
    { current qos++; }

  return 0;



/********************* CURRENT ************************/

int QoS Mapper::Current_Throughput()
```

```
if ((data_table[0]    NULL) II (data_table[1] == NULL))
  return 0;

double acceptable_percentage = 0.10; // if the throughput is within an
// acceptable_percentage of the current QoS (stored as
// current qos) then QoS is increased

unsigned int previous qos   current qos;
unsigned int current_bw = 0;
unsigned int current pos = 0;

// The last data_item that the QoS was smoothed at
static int last_data_item = 0;

// The number of bytes received in the current interval
static int interval_bytes_received = 0;

unsigned int time;

// to calculate the throughput accurately we must discard all packets
// after renegotiation has occurred until the server begins delivering
// packets at the requested rate
if (delivered_qos    current_qos)

    // in order to determine actual number of bytes received, and therefore
    // accurately measure throughput, we need to add the size of the RTP and IP
    // headers and also the control information bytes

    interval_bytes_received += data_table[current_data_item]->packet_size +
                               RTP HEADER + IP HEADER + CONTROL;

    time = data_table[currentdataitem]->received_timestamp
           data_table[last_data_item]->receivedtimestamp;
    cout << "Time\t: " << time << "\n";
  1
else
  1
    last_data_item = current_data_item;
    time = 0;
    interval_bytes_received = 0;


if (time > 0)
1
  double total_time = time / 1000.0;
  double throughput = interval bytes received / total_time;
  pstat_throughput->Add(throughput);
  current_throughput = throughput;
  pstatqosvalues->Add((double) qostable[currentqos]->bandwidth);

  cout << "Throughput\t: " << throughput << " kbps\n";

  if (throughput > (qos_table[delivered_qos]->bandwidth -
                (cps_table[delivered_gos]->bandwidth * acceptable_percentage)))
      // increase QoS by one group, throughput within acceptable amount
      // in order to increase QoS we must make sure that we haven't
      // already increased it according to the current_qos. Only when the
      // delivered qos becomes equal to the current qos can we increase
      increase in qos++;

      // convert prey qos value into a position
      double previous pos = (log10(gostable[previousgos]->bandwidth /
                           gostable[0]->bandwidth) / log10(2));

      // increase by one QoS group, only if we are not already at max
      if (previous_pos < 4)
          current_pos = int(previouspos + 1.0);
```

```
          else
            current pos = int(previouspos);

          current_bw = int(qos_table[0]->bandwidth * int(pow(2, current pos)));

          int index = 0;
          while (qos_table[index]->bandwidth != current_bw)
            {
              index++;

          current qos = index;
      1
    else
      { // must reduce the QoS to the closest QoS to the measured throughput
          cout << " ****** Reducing QoS ******\n";
         decrease in qos++;

          int index = 0;
          while ((qos_table[index] != NULL) && (qos_table[index]->bandwidth <
                 throughput))

              index++;


          if (index != 0)
            index--; // reduce by one or else we will increase the QoS

          current qos = index;



  // if the throughput was greater than max then we have gone OB's , need to
  // reduce by one
  if (qostable[currentqos] == NULL)
    while (qostable[currentqos] == NULL)
      { current qos -= 1; }

  // Now find the most desired QoS at the current throughput
  while ((gos_table[current_qos + 1] != NULL) &&
         (qos_table[current_qos]->bandwidth ==
          gos_table[current_gos + 1]->bandwidth))
    { current qos++; }

  return 0;
1



/********************** SMOOTHED *********** ******** *****/

int QoS Mapper::Smoothed_Throughput()
{
  if (data_table[0]    NULL)
    return 0;

  double acceptable_percentage = 0.10;
  // if the throughput is within an acceptable_percentage of the current QoS
  // stored as current_qos) then QoS is increased

  unsigned int current_bw = 0;
  unsigned int previous_qos = current qos;
  int current_pos;
  static unsigned int interval_packets_received = 0;

  unsigned int increment = 3000;

  // Smoothed loss variable determines the amount of fluctuation allowed in
  // each renegotiation of the QoS. A value of 2 allows the QoS to increase
```

```
// or decrease by a maximum of 2 places
unsigned int max fluctuation = 2;

// The amount of time that the smoothing should apply to
static unsigned int timespan = increment;

// The maximum number of packets to be received before smoothing
unsigned int max packets = 10;

// The last data_item that the QoS was smoothed at
static int last data item = 0;

// The number of bytes received in the current interval
static int interval_bytes_received = 0;

// Add up the total bytes received and the total time taken to receive them
unsigned int time;


// we need to omit the first packet after QoS is renegotiated
if (delivered_qos == current_qos)

    // in order to determine actual number of bytes received, and therefore
    // accurately measure throughput, we need to add the size of the RTP and IP
    // headers and also the control information bytes

    interval_bytes_received += data_table[current_data_item]->packet_size +
                               RTP HEADER + IP HEADER + CONTROL;

    time = data_table[current_data_item]->received_timestamp -
           data_table[lastdataitem]->receivedtimestamp;
    cout « "Time\t: " « time « "\n";

    interval_packets_received++;
  }
else

    last_data_item = current_data_item;
    time = 0;
    interval bytes received = 0;
    interval_packets_received = 0;


if (time > increment)

  double total_time = time / 1000.0;
  double throughput = interval bytes received / total_time;
  pstat_throughput->Add(throughput);
  current_throughput = throughput;
  timespan += increment;
  cout << "Throughput\t: " << throughput << " kbps\n";
  pstatqosvalues->Add((double) clostable[currentqos]->bandwidth);

  if (throughput > (qos_table[delivered_qos]->bandwidth -
                   (qos_table[delivered_qos]->bandwidth *
                    acceptable_percentage)))
    { // increase QoS by one group, throughput within acceptable amount
        cout << "********** Increasing QoS **********\n";
        increase in qos++;

        // convert prey qos value into a position
        double previous_pos = (log10(qos_table[previous_qos]->bandwidth /
                             qostable[0]->bandwidth) / 1og10(2));

        // increase by max fluctuation QoS groups
        if (int(previous_pos + max_fluctuation) > 4)
          current_pos = 4;
        else
```

```
        current pos = int(previouspos + max fluctuation);

      current_bw = int(gostable[0]->bandwidth * int(pow(2, currentpos)));

      int index = 0;
      while (gos_table(index}->bandwidth != current bw)

          index++;

      current qos = index;

else
  { // need to reduce QoS to measured amount
      cout << "********** Decreasing QoS *********\n";
      decreaseingos++;

      // need to measure the actual throughput and then reduce to either
      // the measured value or if it is less than the max_allowed then reduce
      // to max_allowed

      // convert prey qos value into a position
      double previous_pos = (log10(gos_table[previous_gos]->bandwidth /
                          gostable[0]->bandwidth) / log10(2));

      // calculate the closest value in the Qos table to the measured
      // throughput
      int index = 0;
      while (gos_table[index]->bandwidth < throughput)
        { index++; }
      // need to reduce by one or else the QoS is above the throughput

      unsigned int measured_gos;
      if (index > 0)
        measured_gos = index - 1;
      else
        measured qos = 0;

      // convert measured qos into a position
      double measured_pos = (log10(gos_table[measured_gos]->bandwidth /
                          gostable[0]->bandwidth) / log10(2));

      // decrease by max_fluctuation groups
      current_pos = (int)measured_pos;
      if (current_pos < int(previous_pos - max_fluctuation))
        current_pos = int(previous_pos - max_fluctuation);

      if (current_pos < 0)
        current pos = 0;

      current_bw = int(gostable[0]->bandwidth * int(pow(2, currentpos)));

      index = 0;
      while (gos_table[index]->bandwidth != current_bw)
        { index++; }
      current qos = index;


// if the throughput was greater than max then we have gone ob's   need to
// reduce by one
if (gos_table[current_gos] == NULL)
  while (gos_table[current_gos] == NULL)
    { current qos -= 1; }

// Now find the most desired QoS at the current throughput
while ((gos_table[current_gos + 1] != NULL) &&
       (gos_table[current_gos]->bandwidth ==
        gos_table[current_gos + 1]->bandwidth))
  { current qos++; }
```

```
    double packet_rate = (double)interval_packets_received / (double)total_time;
    pstat packetrate->Add(packetrate);

    interval packets received - 0;
    interval bytes received = 0;
    last_data_item = current data item;

  return 0;


int QoS_Mapper;:Smoothed_Packet_Loss()
{
  if (data_table[0] == NULL)
    return 0;

  double minimum_loss = 0.02;
  double maximum_loss = 0.10;
  double loss = 0.0;
  unsigned int previous qos   current qos;
  unsigned int current pos;
  static unsigned int interval_packet_loss = 0;
  static unsigned int previous_packet_loss = 0;
  static unsigned int interval_packets_received = 0;

  unsigned int timespan = 3000;

  // Smoothed loss variable determines the amount of fluctuation allowed in
  // each renegotiation of the QoS. A value of 2 allows the QoS to increase
  // or decrease by a maximum of 2 places
  unsigned int max fluctuation = 2;

  // The maximum number of packets to be received before smoothing
  // unsigned int max packets = 10;

  // The last data_item that the QoS was smoothed at
  static int last_data_item = 0;
  unsigned int current_bw = 0;
  unsigned int time;

  // The number of bytes received in the current interval
  static int interval_bytes_received = 0;

  // we need to omit the first packet after QoS is renegotiated
  if (delivered_gos == current_gos)
    {
      interval_bytes_received += data_table[current_data_item]->packet_size
                                 + RTP_HEADER + IP_HEADER + CONTROL;
      interval_packet_loss = total_packet_loss - previous_packet_loss;
      cout << "Interval loss " << interval_packet_loss << "\n";
      interval_packets_received++;
      time = data_table[current_data_item]¯>received_timestamp -
             datatable[last_data_item]->received_timestamp;
      cout << "Time\t: " << time << "\n";
    }
  else

      cout << "****** Waiting for QoS to change ******\n";
      last_data_item = current_data_item;
      interval packet loss = 0;
      interval_packets_received = 0;
      interval_bytes_received  O;
      previous_packet_loss total_packet_loss;
      time = 0;
      return 0;
```

```
if (time > timespan)

  double total_time = time / 1000.0;
  // Calculate the percentage of lost packets
  loss = (double)interval_packetloss / (double)interval_packets_received;
  pstat_loss->Add(loss);
  current_loss = loss;
  cout << "Packet Loss\t: " << loss <<"\n";
  double throughput = interval bytes received / total_time;
  pstat_throughput->Add(throughput);
  pstatqosvalues->Add((double) gostable[currentgos]->bandwidth);

  // Check if we need to increase the QoS
  if (loss < minimum_loss)

    increaseingos++;

    // convert prey qos value into a position
    double previous_pos = (log10(gos_table[previous_qosj->bandwidth
                         qostable[0]->bandwidth) / log10(2));

    // check if already at minimum QoS, reduce by max fluctuation
    if ((previous pos + max_fluctuation) < 4)
      current_pos = int(previous_pos + max_fluctuation);
    else
      current_pos = 4;
    cout << "current pos " << current pos << "\n";

    currentbw = int(qostable[0]->bandwidth * int(pow(2, current pos)));

    int index = 0;
    while (gos_table[index]->bandwidth != current_bw)

        index++;
      1
    current qos = index;


  // Check if we need to increase the QoS
  if (loss > maximum loss)

    decreaseingos++;

    // convert prey qos value into a position
    double previous_pos = (log10(gos_table[preyious_gos]->bandwidth /
                         qos_table[0]->bandwidth) / 1og10(2));
    cout << "previous pos " << previous pos << "\n";

    if ((previous_pos   max_fluctuation) > 0)
      current_pos = int(preyious_pos - max fluctuation);
    else
      current_pos = 0;
    cout << "current pos " << current pos << "\n";

    current_bw = int(qos_table[0]->bandwidth * int(pow(2, current_pos)));
    cout << "current bw " << currentbw << "\n";

    int index = 0;
    while (gos_table[index]->bandwidth != current_bw)

        index++;
      1
    current qos = index;

    cout << "Current QoS \t: " <<= current qos << "\n";


 double packet rate = (double)interyalpacketsreceived / (double)total time;
```

```
    pstatpacketrate->Add(packetrate);

    last_data_item = current_dataitem;
    interval_bytes_received = 0;
    interval_packet_loss = 0;
    interval_packets_received = 0;
    previouspacket_loss = total_packet_loss;
    time = 0;


  // make sure we don;t go ob's , bring it down if too much
  if (clos_table[current_clos] == NULL)
    while (clos_table[current_clos] == NULL)
      { current qos -= 1; }

  // Now find the most desired QoS at the current throughput
  while ((qos_table[current_gos + 1] != NULL) &&
          (clos_table[current_clos]->bandwidth =-
           qos_table[current_olos + 1]->bandwidth))
    { current qos++;

  return 0;



/******* ***** ********** CONGESTED **********************/

int QoS Mapper::Congested_Throughput()

  if (data_table[0] == NULL)
    return 0;

  // congested, loaded, unloaded are set as the limits (defined in bytes)
  // that the system is in a particular state
  unsigned int congested = 300;
  unsigned int unloaded = 7400;

  // Smoothing loaded and unloaded determine the number of QoS fluctuations
  // that can take place while the system is in a particular stale, i.e. a
  // value of 2 means that the bandwidth can change by two values. i.e. if
  // the value was 88100 then it can decrease to either 44100 or 22050 or
  // increase to 176400 (max value). The higher the value the more
  // fluctuation allowed. The smooth time is the period that the smoothing
  // should apply to

  unsigned int smoothing_congested = 4;
  unsigned int smoothing_loaded = 2;
  unsigned int smoothing_unloaded = 1;
  unsigned int smooth_time = 10000;
  unsigned int max fluctuation = 0;

  // if the throughput is within an
  // acceptable_percentage of the current QoS (stored as
  // current_gos) then QoS is increased
  double acceptable_percentage = 0.10;

  // The previous bandwidth value. This is necessary to endure that the
  // smoothing variables do not adjust the bandwidth too greatly
  unsigned int previous qos = current qos;

  unsigned int current_bw = 0;
  int current_pos = 0;
  unsigned int time;

  // The amount of time that the smoothing should apply to
  unsigned int increment = 3000;
  static unsigned int timespan = increment;
```

```
// The maximum number of packets to be received before smoothing
unsigned int max packets - 10;

// The last data_item that the QoS was smoothed at
static int last data item = 0;

// number of bytes received in current interval
static int interval_bytes_received = 0;

static unsigned int interval_packets_received = 0;

// we need to wait until the control information has been updated on the
// server and packets begin arriving at the requested rate
if (delivered qos    current_qos)

    // in order to determine actual number of bytes received, and therefore
    // accurately measure throughput, we need to add the size of the RTP and IP
    // headers and also the control information bytes

    interval_bytes_received += data_table[current_data_item]->packet_size +
                               RTP HEADER + IP HEADER + CONTROL;

    time = data_table[current_data_item]->received_timestamp -
           datatable[last_data_item]->receivedtimestamp;

    interval packets received++;

else

    lastdata_item current_data_item;
    time= 0;
    interval_bytes_received = 0;
    interval_packets_received = 0;



if (time > increment)

  double total_time = time / 1000.0;
  double throughput = interval_bytes_received / total_time;
  pstat_throughput->Add(throughput);
  current_throughput = throughput;
  timespan += increment;
  cout << "Throughput\t: " << throughput << " kbps\n";
  pstat_qos_values->Add((double) gos_table[current_gos]->bandwidth);

  // Determine the current system state
  if (throughput <= congested)

      max_fluctuation = smoothing_congested;
      cout << "State is congested, fluctuation is " << max_fluctuation «"\n";

  else if (throughput >= unloaded)

      max_fluctuation = smoothing_unloaded;
      cout << "State is unloaded, fluctuation is " << max_fluctuation <<"\n";
    1
  else
    {
      max_fluctuation = smoothing_loaded;
      cout << "State is loaded, fluctuation is " << max_fluctuation << "\n";


  if (throughput > (qos_table[delivered_qos]->bandwidth -
                   (qos_table[delivered_qos]->bandwidth *
                    acceptable_percentage)))
    { // increase QoS by max fluctuation groups, throughput within acceptable
      // amount
```

```
      cout << "********** Increasing QoS **********\n";
      increaseingos++;

      // convert prey qos value into a position
      double previous_pos = (log10(gos_table[previous_gos]->bandwidth /
                          gostable[0]->bandwidth) / log10(2));

      // increase by max fluctuation QoS groups
      if (int(previous_pos + max_fluctuation) > 4)
        current_pos = 4;
      else
        current pos = int(previouspos + max_fluctuation);

      currentbw = int(gostable[0]->bandwidth * int(pow(2, currentpos)));

      int index = 0;
      while (gos_table[index]->bandwidth != current_bw)

          index++;
        1
      current qos = index;

else
  1
      cout << "********** Decreasing QoS *********\n";
      decrease in qos++;

      // convert prey qos value into a position
      double previous_pos = (log10(gos_table[previous_gos]->bandwidth /
                          gostable[0]->bandwidth) / log10(2));

      int index = 0;
      while (gos_table[index]->bandwidth < throughput)
        { index++; }

      // need to reduce by one or else the QoS is above the throughput
      unsigned int measured_gos;
      if (index > 0)
        measured_gos = index - 1;
      else
        measured_gos = 0;

      // convert measured qos into a position
      double measured_pos = (log10(gos_table[measured_gos]->bandwidth /
                          gostable[0]->bandwidth) / log10(2));

      // decrease by max_fluctuation groups
      current_pos = (int)measured_pos;
      if (current_pos < int(previous_pos - max_fluctuation))
        current pos = int(previouspos - max_fluctuation);

      if (current_pos < 0)
        current pos = 0;

      current_bw = int(gos_table[0]->bandwidth * int(pow(2, current_pos)));

      index = 0;
      while (gos_table[index]->bandwidth != currentbw)
        { index++;
      current qos = index;


// if the throughput was greater than max then we have gone ob's , need to
// reduce by one
if (gos_table[current_gos] == NULL)
  while (gostable[currentgos] == NULL)
    { current qos -= 1; }
```

```
    // Now find the most desired QoS at the current throughput
    while ((qos_table[current_clos + 1] != NULL) &&
           (clos_table[current_gos]->bandwidth --
            qos_table[current_clos + 1]->bandwidth))
      { current qos++; }

    double packet_rate = (double)interval_packets_received / (double)total_time;
    pstatpacketrate->Add(packet_rate);

    interval packets received = 0;
    last_data_item = current_data_item;
    interval_bytes_received - 0;
  }
  return 0;


int QoS Mapper::Congested_Packet_Loss()
{
  if (data_table[0] == NULL)
    return 0;

  // congested, loaded, unloaded are set as the limits (defined in bytes)
  // that the system is in a particular state, since this a lookup into the
  // main QoS Table, it can remain in bytes
  double congested = 0.3;
  double unloaded = 0.05;

  // Smoothing loaded and unloaded determine the number of QoS fluctuations
  // that can take place while the system is in a particular state, i.e. a
  // value of 2 means that the bandwidth can change by two values. i.e. if
  // the value was 88100 then it can decrease to either 44100 or 22050 or
  // increase to 176400 (max value). The higher the value the more
  // fluctuation allowed. The smooth time is the period that the smoothing
  // should apply to

  unsigned int smoothing_congested = 4;
  unsigned int smoothing_loaded = 2;
  unsigned int smoothing_unloaded = 1;
  unsigned int timespan = 3000;

  double minimum_loss = congested;
  double maximum loss = unloaded;

  double loss = 0.0;
  unsigned int previous qos = current qos;
  unsigned int current pos;
  static unsigned int interval_packet_loss = 0;
  static unsigned int previous_packet_loss = 0;
  static unsigned int interval_packets_received = 0;

  // Smoothed loss variable determines the amount of fluctuation allowed in
  // each renegotiation of the QoS. A value of 2 allows the QoS to increase
  // or decrease by a maximum of 2 places
  unsigned int max fluctuation = 2;

  // The maximum number of packets to be received before smoothing
  // unsigned int max packets = 10;

  // The last data_item that the QoS was smoothed at
  static int last_data_item = 0;
  unsigned int current_bw = 0;
  unsigned int time;
  static int interval_bytes_received = 0;

  // we need to omit the first packet after QoS is renegotiated
  if (delivered_clos == current_gos)
    {
```

```
    interval_bytes_received += data_table[current_data_item]->packet_size
                               + RTP HEADER + IPHEADER + CONTROL;
    interval_packet_loss = totalpacket_loss - previous_packet_loss;
    cout << "Interval loss " << interval_packet_loss << "\n";
    interval_packets_received++;
    time = data_table[current_data_item]->received_timestamp -
           datatable[lastdata_item]->receivedtimestamp;
    cout << "Time\t: " << time << "\n";
  1
else

    cout << "****** Waiting for QoS to change    ***\n";
    last_data_item = current_data_item;
    interval_bytes_received = 0;
    interval_packet_loss = 0;
    interval_packets_received = 0;
    previous_packet_loss = total_packet_loss;
    time = 0;
    return 0;



if (time > timespan)

  double total_time = time / 1000.0;
  // Calculate the percentage of lost packets
  loss = (double) interval_packet_loss / (double)interval_packets_received;
  pstat_loss->Add(loss);
  current_loss - loss;
  cout << "Packet Loss\t: " << loss <<"\n";
  double throughput = interval_bytes_received / total_time;
  pstat_throughput->Add(throughput);
  pstat_gos_values->Add((double) gos_table[current_clos]->bandwidth);

  // Determine the current system state
  if (loss <= congested)

      max_fluctuation = smoothing_congested;
      cout << "State is congested, fluctuation is " << max_fluctuation

  else if (loss >= unloaded)
    {
      max_fluctuation = smoothing_unloaded;
      cout << "State is unloaded, fluctuation is " << max_fluctuation <<"\n";
    }
  else

      max_fluctuation = smoothing_ loaded;
      cout << "State is loaded, fluctuation is " << max_fluctuation << "\n";
    1

  // Check if we need to increase the QoS
  if (loss < minimum loss)
  {
    increase in qos++;

    // convert prey qos value into a position
    double previous_pos = (log10(gos_table[previous_gos]->bandwidth /
                          gos_table[°]->bandwidth) / log10(2));
    cout << "previous pos : " << previous_pos << "\n";

    // check if already at minimum QoS, reduce by max fluctuation
    if ((previous_pos + max_fluctuation) < 4)
      current_pos = int(previous_pos + max_fluctuation);
    else
      current_pos = 4;
    cout << "current pos " << current pos << "\n";
```

```
      currentbw = int(gostable[0]->bandwidth * int(pow(2, current_pos)));
      cout << "current bw " << current bw << "\n";

      int index = 0;
      while (gostable[index]->bandwidth != current_bw)
        {
          index++;
        1
      current qos = index;

      cout << "Current QoS \t: " << current_gos << "\n";


    // Check if we need to increase the QoS
    if (loss > maximum loss)

      decrease in qos++;

      // convert prey qos value into a position
      double previous_pos = (log10(gos_table[previous_gos]->bandwidth /
                           gos_table[0]->bandwidth) / log10(2));
      cout << "previous pos " << previous pos << "\n";

      if ((previous_pos - max_ fluctuation) > 0)
        current_pos = int(previous_pos - max_ fluctuation);
      else
        current_pos = 0;
      cout << "current pos " << current pos << "\n";

      current_bw = int(gos_table[0]->bandwidth * int(pow(2, current_pos)));
      cout << "current bw " << current bw << "\n";

      int index = 0;
      while (gos_table[index]->bandwidth != current_bw)
        {
          index++;
        }
      current qos = index;


    double packet_rate = (double)interval_packets_received / (double)total_time;
    pstatpacketrate->Add(packet_rate);

    last_data_item = current_data_item;
    interval bytes received = 0;
    interval_packet_loss = 0;
    interval_packets_received = 0;
    previous_packet_ioss = total_packet_loss;
    time = 0;


  // make sure we don;t go ob's , bring it down if too much
  if (gos_table[current_gos] == NULL)
    while (gos_table[current_gos] == NULL)
      ( current qos -- 1; )

  // Now find the most desired QoS at the current throughput
  while ((gostable[currentgos + 1] != NULL) &&
         (gos_table[current_gos]->bandwidth ==
          gos_table[current_gos + l]->bandwidth))
    { current qos++; }

  return 0;
1
```

```
int QoS Mapper::Adjusted Method()
{
  if (data_table[0] == NULL)
    return 0;

  // congested, loaded, unloaded are set as the limits (defined in bytes)
  // that the system is in a particular state, since this a lookup into the
  // main QoS Table, it can remain in bytes
  double congested = 1000;
  double unloaded = 7400;

  // Smoothing loaded and unloaded determine the number of QoS fluctuations
  // that can take place while the system is in a particular state, i.e. a
  // value of 2 means that the bandwidth can change by two values. i.e. if
  // the value was 88100 then it can decrease to either 44100 or 22050 or
  // increase to 176400 (max value). The higher the value the more
  // fluctuation allowed. The smooth time is the period that the smoothing
  // should apply to

  unsigned int smoothing_congested = 1;
  unsigned int smoothing loaded = 2;
  unsigned int smoothing_unloaded = 4;
  unsigned int timespan = 3000;

  double minimum_loss = 0.1;
  double maximum loss = 0.2;

  // if the throughput is within an
  // acceptable_percentage of the current QoS (stored as
  // current_qos) then QoS is increased
  double acceptable_increase = 0.10;
  double acceptable decrease = 0.25;

  double loss = 0.0;
  unsigned int previous qos = current qos;
  unsigned int current pos;
  static unsigned int interval_packet_loss = 0;
  static unsigned int previous_packet_loss = 0;
  static unsigned int interval_packets_received = 0;

  // Smoothed loss variable determines the amount of fluctuation allowed in
  // each renegotiation of the QoS. A value of 2 allows the QoS to increase
  // or decrease by a maximum of 2 places
  unsigned int max fluctuation = 2;

  // The maximum number of packets to be received before smoothing
  // unsigned int max packets = 10;

  // The last data_item that the QoS was smoothed at
  static int last_data_item = 0;
  unsigned int current_bw = 0;
  unsigned int time;
  static int interval_bytes_received = 0;

  // we need to omit the first packet after QoS is renegotiated
  if (delivered_qos == current_qos)
    {
      interval_bytes_received += data_table[current_data_item]->packet_size
                                 + RTP_HEADER + IP_HEADER + CONTSOL;
      interval_packet_loss = totalpacket_loss - previous_packet_loss;
      cout << "Interval loss " << interval_packet_loss « "\n";
      interval_packets_received++;
      time = data_table[current_data_item]->received_timestamp
             data_table[last_data_item]->received_timestamp;
      cout << "Time\t: " << time << "\n";
    }
  else
```

```
      cout << "****** Waiting for QoS to change  ******\wv ;
      last_data_item = current_data_item;
      interval_bytes_received = 0;
      interval_packet_loss = 0;
      interval_packets_received = 0;
      previous packet loss = total packet loss;
      time = 0;
      return 0;



if (time > timespan)

   double total_time = time / 1000.0;
   // Calculate the percentage of lost packets
   loss = (double)interval_packet_loss / (double)interval_packe s received;
   pstat_loss->Add(loss);
   current_loss = loss;
   cout << "Packet Loss\t:  " « loss <<"\n";
   double throughput = interval_bytes_received / total_time;
   pstat_throughput->Add(throughput);
   pstatgosvalues->Add((double) gostable[currentgos]->bandw'dth);

   // Determine the current system state
   if (throughput <= congested)

        max_fluctuation = smoothing_congested;
        cout << "State is congested, fluctuation is " << max_fluctuation <<"\n";
     ]
   else if (throughput >= unloaded)

        max_fluctuation = smoothing unloaded;
        cout << "State is unloaded, fluctuation is " << max fluc uation <<"\n";

   else
     {
        max_fluctuation smoothing_loaded;
        cout << "State is loaded, fluctuation is " << max_fluctuation << "\n";
     }

   // Check if we need to increase the QoS
   if ((loss < minimum loss) && (throughput >
         (clos_tablefdelivered_clos]->bandwidth -
         (clos_table[delivered_clos]->bandwidth * acceptable_increase))))


     increase in qos++;

     // convert prey qos value into a position
     double previous pos = (1og10(gos_table[previous_gos]->band idth /
                                  clos_table[0]->bandwidth) / log10(2));
     cout << "previous pos : " << previous pos << "\n";

     // check if already at minimum QoS, reduce by max fluctuat on
     if ((previous_pos + 1) < 4)
        current_pos = int(previous_pos + 1);
     else
        current_pos = 4;
     cout << "current pos " << current pos << "\n";

     current_bw = int(clos_table[0]->bandwidth * int(pow(2, curr nt_pos)));
     cout << "current bw " << current_bw << "\n";

     int index - 0;
     while (clos_table[index]->bandwidth != current_bw)

          index++;
```

```
    current qos = index;

    cout << "Current QoS \t: " << current qos << "\n";


  // Check if we need to reduce the QoS
  if (loss > maximum loss)
  if ((loss > maximum_loss) II (throughput <
      (clos_table[delivered_clos]->bandwidth -
      (qostable[deliveredgos]->bandwidth * acceptable_decrease))))


    decrease in qos++;

    // convert prey qos value into a position
    double previous_pos = (log10(qos_table[previous_qos]->bandwidth /
                         closlable[0]->bandwidth) / log10(2));

    if ((previous_pos - max_ fluctuation) > 0)
      current_pos = int(previous_pos - max_ fluctuation);
    else
      current pos = 0;

    currentbw = int(clostable[0]->bandwidth * int(pow(2, current_pos)));

    int index = 0;
    while (gos_table[index]->bandwidth != current_bw)

        index++;
      1
    current qos = index;

  1
  double packet_rate = (double)interval_packets_received / (double)total_time;
  pstat_packet_rate->Add(packet_rate);

  last_data_item = current_data_item;
  interval_bytes_received = 0;
  interval_packet_loss = 0;
  interval_packets_received = 0;
  previous_packet_loss   total_packet_loss;
  time = 0;


// make sure we don;t go ob's , bring it down if too much
if (gos_table[current_clos]    NULL)
  while (clos_table[current_clos] == NULL)
    { current qos    1; }

// Now find the most desired QoS at the current throughput
while ((qostable[currentgos + 1] != NULL) &&
       (clos_table[current_clos]->bandwidth ==
        clos table[currentclos + 1]->bandwidth))
  { current qos++; }

return 0;
1
```

References

1. Formal Literature

[ATM 1994]

The ATM-Forum, *ATM User-Network Interface Specification, v.3.1,* Sept. 1994

[Alfano 1995]

Alfano, M., *A Quali0 of Service Management Architecture (QoS1V171): A Preliminary Study,* Berkerley, December 1995.

[Alfano and Radouniklis 1996]

Alfano, M., and Radouniklis, N., *A Cooperative Multimedia Environment with QoS Control: Architectural and Implementation Issues,* International Computer Science Institute, Berkerley, September 1996.

[Aurrecoechea *et. al.]*

Aurrecoechea, C., Campbell, A. and Hauw, L., *A Survey of ,QoS Architectures,* Columbia University.

[Banerjea *et. al.* 1996]

Banerjea, A., Ferrari, D., Mah, B.A., Moran, M., Verma, D. C., and Zhang, H., *The TENET Real-Time Protocol Suite : Design, Implementation, and Experiences,* February 1996.

[Barth *et. al.]*

Barth, I., Dernaler, G.,Fiederer, W., and Rothermel, K., *A Framework for°salable Quality of Service in Distributed Multimedia Systems,* University of Stuttgart.

[Basso *et.al.]*

Basso, A., Cash, G. and Civanlar, M., *Transmission of MPEG-2 Streams over Non-Guaranteed ,Quali0 of Service Networls,* AT&T Labs — Research.

[Berra *et. al.* 1992]

Berra, P.B., Chen, C-Y. R., Ghafoor, A., and Little, T.D.C., *Issues in Networking and Data Management of Distributed Multimedia Systems,* Symposium on High-Performance Distributed Computing, New York, September 1992.

[Bolot et. al.]

Bolot, J-C, Crepin, H., and Garcia, A., *Analysis of Audio Packet Loss in the Internet,* INRIA, France.

[Bradner 1997]

Bradner, S., *Internet Protocol Qualio of Service Problem Statement,* Network Working Group, Internet-Draft, September 1997.

[Burk and Horvath 1997]

Burk, R., and Horvath, D., *UNIX Unleashed, System Administrator's Edition,* Macmillan Computer Publishing, Indianapolis, 1997.

[Busse *et. al.* 1995]

Busse, I., Deffner, B., and Schulzrinne, H., *Dynamic ,QoS Control of Multimedia Applications based on* RTP, May 1995.

[Campbell and Coulson]

Campbell, A.T., and Coulson, G., *QoS Adaptive Transports: Delivering Scalable Media to the Desktop,* Lancaster University.

[Campbell *et. al.* 1993]

Campbell, A., Coulson, G., Garcia, F., Hutchinson, D., and Leopold, H., *Integrated Quality of Service for Multimedia Communications,* IEEE Infocom'93, San Francisco, March 1993.

[Campbell *et. al.* (1)]

Campbell, A., Coulson, G., and Hutchinson, D., *A Multimedia Enhanced Transport Service in a Qualiy of Service Architecture,* Lancaster University.

[Campbell *et. al.* (2)]

Campbell, A., Coulson, G., and Hutchinson, D., *SupportingAdaptive Flows in a ,QualOi of Service Architecture,* Lancaster University.

[Campbell *et. al.* (3)]

Campbell, A., Hutchinson, D., and Aurrecoechea, C., *Dynamic QoS Management for Scalable Video Flows,* Columbia University.

[Comer 1991]

Corner, D., *Internetworking with TCP/IP,* Prentice-Hall, 1991.

[Coulson *et. al.* 1994]

Coulson, G., Campbell, A., Robin, P., Blair, G., Papathomas, M., and Hutchinson, D., *The Design of a QoS Controlled ATM based Communications System in Chorus,* Lancaster University, 1994.

[Deering and Hinden 1995]

Deering, S., and Hinden, R., *Internet Protocol, Version 6 (IPv6) Specification,* RFC 1883, Network Working Group, December 1995.

[Dermler *et. al.* 1995]

Dermler, G., Fiederer, W., Barth, I., and Rothermel, K., *A Framework for Negotiable Quality of Service in Distributed Multimedia Systems,* University of Stuttgart, December 1995.

[Dermler *et. al.]*

Dermler, G., Fiederer, W., Barth, I., and Rothermel, K, *A Negotiation and Resource Reservation Protocol (NRP) for Configurable Multimedia Applications,* University of Stuttgart.

[Ellis and Stroustrup 1991]

Ellis, M. A., and Stroustrup, B., *The Annotated C++ Reference Manual,* Addison-Wesley Publishing Company, December 1991.

[ISI 1981]

Information Sciences Institute, *Internet Protocol — DARPA Internet Program Protocol Specification,* RFC 791, September 1981.

[Littlejohn and Clayton 1997]

Littlejohn, P.S., and Clayton, P., *Quality of Service Issues in the Delivery of Multimedia Content,* Proc. Teletraffic '98, Grahamstown, South Africa, September 1997.

[Littlejohn and Clayton 1998]

Littlejohn, P.S., and Clayton, P., *Adaptive Flow Management of Multimedia Da'a with a Variable Quality of Service using the Real-Time Protocol,* Proc. SA Telecommunications, etworks & Applications Conference, Cape Town, South Africa, September 1998.

[Mankin *et. al.* 1997]

Mankin, A., Baker, F., Braden, B., Bradner, S., O'Dell, M., Romanow, M., Weinrib, A., and Zhang, L., *Resource ReSerVation Protocol (RSVP) Version 1 Applicability tatement,* RFC 2208, Network Working Group, September 1997.

[Parker 1996]

Parker, T., *Teach Yourseif TCP/IP in 14 Days,* Second Edition, Sams Publis ing, Indianapolis, 1996.

[Rothermel *et. al.* 1996]

Rothermel, K., Dermler, G., and Fiederer, W., *„QoS Negotiation and Resource Reservation for Distributed Multimedia Applications,* University of Stuttgart, July 1996.

[Schulzrinne 1994]

Schulzrinne, H., *Audio and Video over Packet Networks Issues, Architecture a d Protocols,* Interop'94 Paris, October 1994.

[Schulzrinne 1996]

Schulzrinne, H., *RIP Profile for Audio and Video Conferences with Minimal Co trol,* RFC 1890, Network Working Group, January 1996.

[Schulzrinne *et. al.* 1996]

Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V., R./ *P: A Tran ort Protocol for Real-Time Applications,* RFC 1889, Network Working Group, January 1996.

[Schulzrinne *et. al.* 1998]

Schulzrinne, H., Rao, A., and Lanphier, R., *Real-Time Streaming Protocol (R SP),* RFC 2326, Network Working Group, April 1998.

[Shenker *et. al.* 1997]

Shenker, S., Partridge, C., and Guerin, R., *Specification of Guaranteed Quality if Service,* RFC 2212, Network Working Group, September 1997.

[Tranter 1996]

Tranter, J., *Linux Multimedia Guide,* O'Reilly & Associates, California, Sept mber 1996.

[Zhang *et. al.]*

Zhang, L., Deering, S., Estrin, D., Shenker, S., and Zappala, D., *RSVP: A New Resource Reservation Protocol.*

2. Online References

[4Front]

4Front Technologies, *OSS Programmers Guide,* Online: http: //www. 4 frOnt - tech . com/index . html

[CNR]

CNR IASI Netlab, RI *P Library,* Online: http://ultral.iasi.rm.cnr.it/welcome.html

[Icast]

Icast Corporation, *Mbone Information IVeb,* Online: http: / /www.mbone . corn/

[Ramaswamy]

Ramaswamy, A., *Understanding MPEG for Video Compression,* Vela Research Inc., Online: http://www.vela.com/-arun/COMP.html.

[Rubenstein 1997]

Rubenstein, D., *Columbia RIP Library API Specification,* Columbia University, Online: http: //gaia.cs.umass.edu/-drubenst/rtp_api.html,Augut 1997.