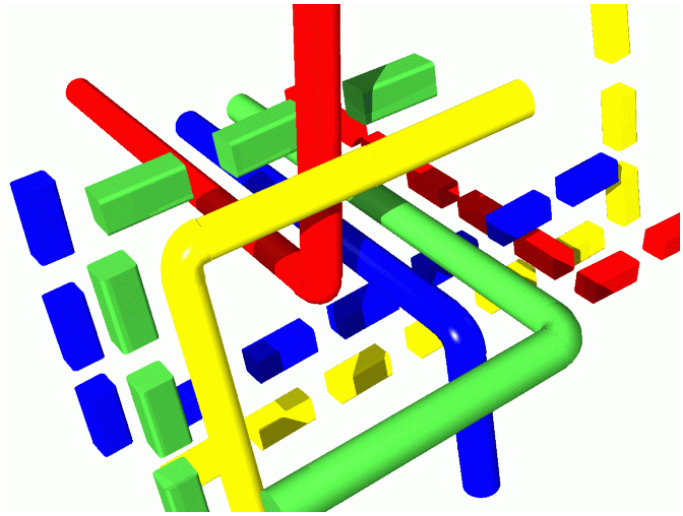
QoS based Real-Time Audio Streaming in the Internet

Diplomarbeit an der Universität Ulm
Fakultät für Informatik



vorgelegt von:

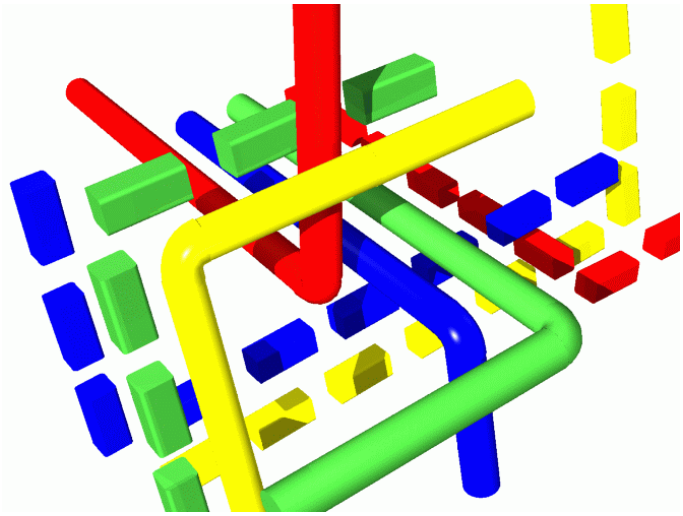
Stefan Schmid

1. Gutachter: *Dr. habil. Konrad Froitzheim*
2. Gutachter: *Prof. David Hutchison*

1999

QoS based Real-Time Audio Streaming in the Internet

Master Thesis at the Universität Ulm
Fakultät für Informatik



submitted by:

Stefan Schmid

1. Examiner: *Dr. habil. Konrad Froitzheim*
2. Examiner: *Prof. David Hutchison*

1999

Abstract

Live audio streaming is an important component of Internet multimedia. Today's Internet, however, offers only poor support for such streams due to the lack of Quality of Service (QoS) capabilities.

The work presented in this thesis discusses the problems of real-time audio streaming and investigates solutions for improving the QoS provided in current and next generation IP networks.

The contributions of this work can be divided into three parts:

First, the thesis provides an in-depth discussion on important Internet multimedia protocols and mechanisms intended to improve the QoS of real-time audio streaming in the Internet. The study evaluates these protocols and mechanisms for use within (interactive) real-time streaming applications and highlights a set of mandatory and recommended techniques. Current network level QoS mechanisms, especially DiffServ and IntServ, are compared and examined with respect to their prospective use in a future QoS framework for the Internet.

Second, WebAudio, a state-of-the-art real-time audio streaming application, is introduced. This uni-directional audio streaming application is designed in a platform independent fashion. The "open" stream control interface enables easy Web integration. The WebAudio server and client applications improve the streaming QoS by means of resource reservation and adaptation. The multi-streaming capabilities of WebAudio allow the applications to be used for audio conferencing. An extensive discussion on the application architecture and the implementation issues of the WebAudio server and client is presented.

Third, the thesis explores the benefits of IPv6 for network QoS mechanisms such as IntServ/RSVP. This work questions whether there is an efficiency gain in packet classification due to the employment of network level flow labels. Based on WebAudio and on a flow label-enabled and extended RSVP implementation, a series of experiments has been performed. The results indicate that flow label based packet classification performs of the order of 2-4 times better than standard IPv4 classification, and it outperforms standard IPv6 classification by about 3-6 times.

The combined results of the work presented in this thesis make a strong contribution towards understanding how to improve QoS for future multimedia applications running over the Internet.

Acknowledgments

The author would especially like to thank his supervisors Dr. Konrad Froitzheim and Professor David Hutchison for their initial inspiration and on-going guidance.

I am grateful for the assistance received from my colleagues in the Distributed Multimedia Research Group at Lancaster and from other friends. A special vote of thanks should be made to Andrew Scott and Laurent Mathy, who have commented on the work and contributed in many ways. Katia Saikoski, Joe Finney and Albrecht Schmidt must also be credited for reading drafts of the thesis.

Thanks are also due to BT Labs, who offered remote access to their Futures Testbed for experimental use.

I would also like to express great appreciation to my family and other friends not mentioned here who continued to provide support and encouragement.

Finally, and most especially I would like to thank my friend Katrin Michlmayr for reading the thesis and supporting me with encouragement and care throughout this time.

Lancaster, April 1999 Stefan Schmid

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multimedia streaming applications, also known as continuous media applications, have become increasingly popular in the Internet. There are several factors responsible for this development, however, three driving forces behind this growth are especially noteworthy. First, today's end-user desktop machines already have extensive multimedia facilities such as audio and video support built-in (for example, sound cards, frame grabbers, hardware and software coders). At present time users are accustomed to applications that exploit those multimedia capabilities and are rather disappointed by software that does not provide multimedia functionality. Secondly, current Internet technology, including new protocols specifically designed for multimedia streaming (for example, RTP, RSVP and RTSP) and experimental multimedia software[1] is becoming ready for deployment. Third, the rapidly growing popularity of the Internet and in particular the *World Wide Web (WWW)* tempt many users to explore novel online services and capabilities. Together, these forces explain the increasing interest in media streaming applications for the Internet.

Continuous media applications can be divided into those that stream pre-recorded continuous media data stored on so called media servers and those that deliver real-time or live media. The first group is simply called *continuous (media) streaming applications*. Some examples are applications for audio and video on-demand or distance learning based on streaming audio and video clips. A second group is called *real-time (media) streaming applications*. This group distinguishes applications that have live data feeds rather than stored media and is currently of particular public interest driven by the idea of *Internet Telephony*, also referred to *Voice over IP (VoIP)*.

Recent research in this area has attracted a number of commercial companies to develop hardware and software based products for VoIP, which they hope will replace current telephone systems. Many believe that the two main advantages of VoIP over the *Public Switched Telephone Network (PSTN)* are: First, VoIP enables flexible telephony applications with support for mobility, redirection, and absence that are simply based on user

[1] Examples include Berkeley's vic [JMat] and vat [MJ95], UCL's rat [H+95], GMD Fokus's NetVoT [Sch92], and INRIA's FreePhone [BVGFPne].

applications for desktop computers. Secondly that VoIP has the potential to reduce current call costs (audio compression allows higher utilization of connections). However, this has not yet been proven in large scale experiments, it remains unclear what impact full accounting would have on VoIP systems.

Other examples of real-time streaming applications, which have been available for some time, are multimedia conferencing, distributed workshops and tele-teaching, interactive multimedia games and radio-broadcasting.

The Internet, which is supposed to provide the common internetwork infrastructure for current "data transfer" applications and those "new" and very "different" applications, has only limited support for real-time applications and applications with high demands for *Quality of Service (QoS)* (see section 1.2).

The Internet was originally designed for simple data transfer, such as message exchange (Email), file transfer (FTP) or remote access (Telnet) among inter-connected computers. These applications have restricted resource demands and/or loose time requirements. This led to a very simple and scalable design of the network that offers *best-effort* service, in which the network does not guarantee anything. This service model was chosen mainly due to its simplicity. Since most early applications using the network could cope with a wide range of underlying service quality, a simple *best-effort* service was appropriate. The simplicity of this *best-effort* approach has undoubtedly contributed to the wide scaled deployment of the Internet.

Over time the Internet has become a victim of its own success. In the beginning, it was mainly known as a military and research network. Later, in the nineties, the WWW attracted many users as convenient information service. And now, users, still fascinated by the extensive opportunities of the world-wide internetwork, are inspired by the idea of using the Internet for real-time audio and video communication or video-on-demand applications. As a result of this fast development, large sections of the Internet are often heavily overloaded. The simple *best-effort* service which shares the bandwidth fairly among all users leads the network into congestion. This results in increased delay variations, called jitter, and packet loss.

With regard to network congestion, real-time streaming applications contribute heavily to congestion, because of their large bandwidth requirements, and suffer from it more than other applications. Non-real-time applications simply slow down when congestion occurs since data transfer takes longer to complete and lost packets can be be retransmitted. Real-time applications, in contrast, become unusable under heavy load; real-time data that arrives late is normally obsolete.

The work presented within this thesis aims to contribute to the area of Internet real-time media streaming in several ways. First, it provides a basic analysis of the problems caused by the lack of QoS mechanisms in the current Internet. Second, a study of state-of-the-art Internet multimedia protocols which explores their usability and importance for current Internet multimedia applications are presented. The study outlines some experimental

protocols which aim to improve the QoS offered to real-time streaming applications. Third, the thesis introduces a new real-time audio streaming application called WebAudio (see section 4) to which built-in support for the next generation Internet protocol, namely IPv6 (see section 2.1.1), and QoS based on the *Integrated Services (IntServ)* architecture (see section 3.2.5) and the *Resource reSerVation Protocol (RSVP)* (see section 2.3.1) were added. Fourth, the thesis presents a theoretical and experimental analysis of the benefits of IPv6 flow labels (see section 5.3) for QoS mechanisms based on IntServ and RSVP. And finally, the thesis provides a critical discussion on the resource reservation protocol RSVP and outlines a few extensions.

Before the discussion on the issues of real-time media streaming and QoS is carried on, section 1.1 classifies real-time streaming applications and their media traffic in relation to other Internet applications and traffic characteristics. Section 1.2 explores the common understanding of QoS and discusses what is currently provided in the Internet. Third, section 1.3 discusses the QoS requirements of real-time media streaming application and in particular for audio streaming.

1.1 Traffic Classification

Internet traffic can be divided into two fundamentally different traffic types: *real-time traffic* and *non-real-time traffic* which is often referred to as *data traffic* [B+94] or *discrete data traffic*. Figure 1.1 respectively illustrates a traffic and application classification.

Real-time applications which deliver *continuous* data streams usually have regular (i.e. constant bit, frame or packet rates) and long lasting (i.e. video clips or live audio) traffic characteristics. These *real-time streaming applications* usually process the data as soon as a defined amount, such as a video frame, has arrived. *Real-time applications* which do not process data streams usually produce bursty interactive traffic with very strict end-to-end delay constraints. These applications can be classified as *real-time control applications* (for example, interactive games or remote-machine control).

The characterization of "real-time" is a relative or elastic property. In fact, one could say that all "real-time" applications are only "quasi-real-time" applications. All so-called real-time applications tolerate small delays. In comparison to *elastic applications*, they are more sensitive to delay and delay variations. The important QoS properties for "real-time" communication, namely delay, jitter and loss, depend entirely on the application itself. Applications to remote-control fast machines, for example, have much stricter timing requirements than a video-on-demand application.

Real-time streaming applications with loose time constraints are also referred to as *tolerant real-time streaming applications*. Applications with very strict time requirements, on the other hand, are called *intolerant* or *rigid real-time streaming applications*.

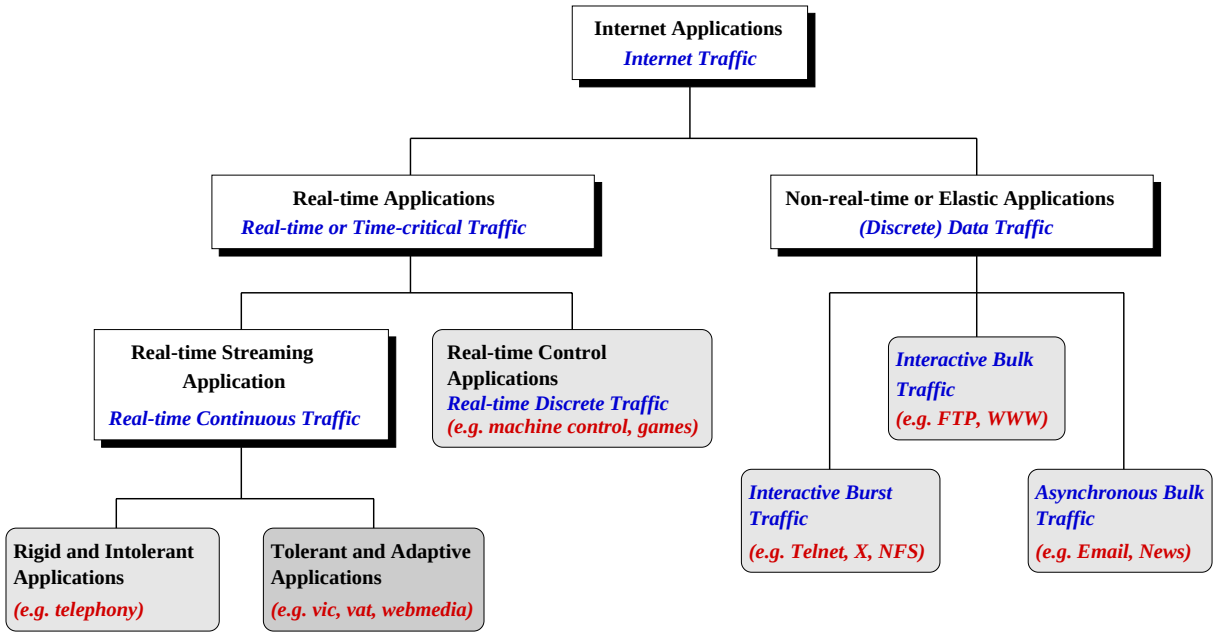


Figure 1.1: Traffic and Application Classification

Applications which have by their nature very strict QoS constraints, such as real-time audio streaming applications (due to the strong sensitivity of the human sound perception), can become *tolerant* to QoS interruptions by means of adaptation (see section 3.1.3). *Rigid applications*, for example, have a fixed playback point, whereas *adaptive applications* are capable of adjusting their playback point according to the observed network QoS.

Elastic applications like Telnet, FTP, WWW, Email, etc. produce *discrete data traffic*, where the individual data packets are sent loosely coupled and without time constraints between each other. These applications usually wait for certain amount of data to arrive, before starting to process them. Therefore, long delays and jitter, as a result of bad network conditions, degrade the performance, but do not affect the final outcome of the data transfer. *Elastic applications* can be further classified according to their delay requirements. *Bulk traffic* (for example, Email, News), asynchronously delivered in the background, operates well even with high transmission delay. *Interactive burst traffic* (for example, Telnet, X, NFS), on the other hand, requires minimal delay to achieve acceptable responsiveness. *Interactive bulk traffic* (for example, FTP, WWW) operates well with medium delays.

(Discrete) Data traffic is known to be of a bursty nature. This is simply due to the fact that *elastic applications* usually send out data as fast as the connection allows. Moreover, these connections are usually very transient. They exist only to transfer one or at most a few packets of data. As a result, *data traffic* is, in general, considered to be unpredictable.

1.2 Quality of Service in the Internet

The usability or the success of continuous multimedia application depends largely on the *Quality of Service (QoS)* they provide to the end users.

As discussed in detail later in section 1.3.1, the quality requirements of real-time audio streaming applications are crucial. Delayed data transfer, even if delayed only fractionally too much, makes two-way communication intolerable. Loss of signals is also instantaneously recognized by human sound perception. Thus, the QoS offered by Internet multimedia applications is an important issue for their usability and success.

The next section brings to light what *Quality of Service* really means to people.

1.2.1 QoS — What is it?

QoS is currently one of the most elusive and confusing topics in the area of data networking. One reason surely comes from the expression itself. Both words, quality and service, are fairly vague and ambiguous. Another reason might be that QoS has different meanings in different contexts or to different people. It is important to understand the different meanings. To some, QoS means introducing an element of predictability and consistency into existing best-effort networks. To others, it means obtaining higher transport efficiency or throughput from the network. And yet to others, QoS is simply a means of differentiating classes of data service. It may also mean to match the allocation of network resources to the characteristics of specific data flows.

To examine the concept of QoS in detail, its two operative words, *quality* and *service*, are first examined.

Quality in networking is commonly used to describe the process of delivering data in a certain manner, sometimes reliable or simply better than normal. It includes the aspect of data loss, minimal delay or latency and delay variations. Determining the most efficient use of network resources, such as the shortest distance or the minimal congested route, is also an issue expressed by quality.

The term *service* has several meanings. It is generally used to describe something offered to end-users of a network. Services can provide a wide range of offerings, from application level services, such as Email, WWW, etc., to network or link level services such as end-to-end connection (for example, TCP connections or ATM virtual channels).

The composition of the terms *quality* and *service* in the context of networking, however, puts forth a fairly straightforward definition: network QoS is a measurement of how well the network operates and a means to define the characteristics of specific network services. Accordingly, the ISO standard defines QoS as a concept for specifying how "good" a networking service is. Therefore, QoS provides the means to evaluate services. For example, *Internet Service Providers (ISP)* provide more or less the same service, except that they usually provide different quality.

1.2.2 QoS Parameters

QoS parameters provide a means of specifying user requirements that may or may not be supported by underlying networks. QoS can only be guaranteed at higher layers if the underlying layers are also able to guarantee this QoS. The QoS values are usually agreed between the service provider and the customer at the time the customer subscribes to a particular service. Here, the customer could be another service provider on a lower level (an instance who uses the service) whereas the service provider offers some form of services. QoS parameters also form a basis for charging customers for pre-specified services.

With the increasing interest in continuous media streaming applications such as audio and video, QoS is becoming more and more important. There are several aspects of QoS to be considered. For example, to support video communication high throughput is required and, therefore, high bandwidth guarantees will have to be made. Audio communication, in contrast, does not usually require high bandwidth. End-to-end delay and delay variations are other factors that must be taken into account for time-critical traffic. In particular, interactive or real-time media streaming communication imposes stringent delay constraints, derived from human perceptual thresholds, which must not be violated. Jitter must also be kept within rigorous bounds to preserve the intelligibility of audio and voice information.

A set of QoS parameters [FH98] suitable for characterizing the quality of service of individual "connections" or "data flows" is as follows:

Delay

> End-to-end transit delay is the elapsed time for a packet to be passed from the sender through the network to the receiver. The higher the delay between the sender and receiver, the more insensitive the feedback loop becomes, and therefore, the protocol becomes less sensitive to short term dynamic changes in the network. For interactive or real-time applications the introduction of delay causes the system to appear unresponsive and as a result in many cases unusable.

Jitter

> The variation in end-to-end transit delay is called jitter, also often referred to as delay variation. In packet-switched networks jitter defines the distortion of the inter-packet arrival times compared to the inter-packet times of the packet transmission. High levels of jitter are unacceptable in situations where the application is real-time. In such cases the distorted data can only be rectified by increasing the receiver's reassembly buffer, which effects the end-to-end delay, making interactive sessions very ponderous to maintain. The strong interconnection between the end-to-end delay and the jitter should be noted. The jitter in the network has a direct impact on the minimum end-to-end delay that can be guaranteed by the network.

Bandwidth

The maximal data transfer rate that can be sustained between two end points of the network is defined as the bandwidth of the network link. It should be noted that the bandwidth is not only limited by the physical infrastructure of the traffic path within the transit networks, which provides an upper bound to the available bandwidth, but is also limited by the number of other flows sharing common resources on this end-to-end path. The term bandwidth is used as an upper bound of the data transfer rate, whereas the expression throughput is used as an instant measurement of the actual exchanged data rate between two entities. Network applications, for example, have a certain bandwidth disposable between two nodes, but the amount of data they really transmit is determined by their throughput. The data throughput of an application is usually highly dynamic, depending on its needs.

$$0 \leq Throughput \leq Bandwidth \tag{1.1}$$

Reliability

This property of the transmission system determines the average error rate of the transit network. The error rate can be subdivided into *bit error rate* and *packet* or *cell error rate*. A poorly configured or poorly performing switching system can reorder packets during transmission, delivering packets to the receiver in a different order than originally transmitted, or even drop packets through transient routing loops. In the case of packet audio or video unreliable networks may induce distortion in the original signal at the receiver's end. Transport level (or higher level) mechanisms are required to detect and correct reordered packets. Usually, re-assembly buffers, which implicitly increase the playback delay and hence decrease the responsiveness, are deployed to rectify disordered packets or to reconstruct lost packets.

The QoS requirements should be negotiated end-to-end at the time of connection or data flow establishment. Preferred, acceptable and unacceptable tolerance levels for each of these QoS parameters should be quantified and expressed. The finally agreed QoS should then be guaranteed for the duration of the transmission or at least an indication must be provided if the contractual values are violated.

1.2.3 Dynamic QoS Control

Because of the increasing demand on QoS requirements, current and future communication architectures (for example, applications, networks, etc.) must be extended to support dynamic QoS selections so that customers are able to precisely tailor individual transport connections to their particular requirements.

It is usually disadvantageous to limit the QoS negotiation at the time of connection establishment. Specified QoS levels do not often remain valid for the lifetime of the transmission.

Hence, dynamic QoS control which allows users to alter the QoS of a connection or data flow during the session is preferential. A user might decide during an audio session to upgrade the audio quality from that of a standard telephone quality to CD quality.

State-of-the-art multimedia applications make use of dynamic QoS control mechanisms to dynamically negotiate their instantaneous QoS demands. The benefits of dynamic QoS control mechanisms are mainly flexibility and cost reduction. First, the application can change its QoS level whenever this is desired rather than having to stick to the initial negotiation. Second, if lower QoS is required, service costs can be reduced by simply degrading the QoS level.

In order to enable QoS (re-)negotiation at the transport level, the necessary support must be provided either within or below the transport protocol. Thus, mechanisms are required to dynamically alter link-level QoS properties on intermediate network nodes. One possibility is to reconfigure the network by means of resource reservation protocols (see section 2.3) where resources are reserved in the intermediate nodes along the transmission path.

1.2.4 QoS within Today's Internet

This section discusses the relation of QoS and the current Internet or, in other words, what form of QoS is supported within today's Internet?

The Internet is composed of a large collection of intermediate network nodes, called routers, and network links connecting the routers. Upon the arrival of a packet, routers determine the next hop interface (the link to the next router or destination) and place the packet in an output queue of the selected interface. The network links have specific QoS characteristics with respect to *delay*, *jitter*, *bandwidth* and *reliability*. The QoS properties of network links depend entirely on the link layer characteristics and physical transmission medium.

If the amount of traffic transmitted on a particular link exceeds the available *bandwidth* of the link for a period of time, congestion occurs that results in poor service quality. In such cases the router's output queue of the saturated transmission link fills up. Variations in the queue length have a direct impact on the *jitter*. They increase the *jitter* and, as a result, the end-to-end *delay*. If high load remains until queues overflow, routers are forced to discard packets which in turn reduces the *reliability*. As a result, packet loss can be used as an identifier for congestion in the network (see section 2.2.2). Adaptive applications or transport protocols (for example, TCP, see section 2.2.2) reduce their sending rate, or the *bandwidth* used by the application, upon detection of network congestion to minimize packet loss.

Poor service quality might also be a result of instabilities in the routing protocol. Such instabilities may cause routers frequently to change their choice of the best next-hop interface, causing data packets of the same flow to take different transmission paths. This bears the risk of out-of-order packet delivery and packet loss reducing *reliability*. Reordering buffers, required to rearrange the original packet ordering, increases *jitter*.

This brief operational overview shows that the Internet, as we know it today, has no explicit support for QoS. Hence, it is rightly called a *best-effort* network lacking of any kind of QoS management or control. Each network element simply tries to do the "*best*" it can. Since there is at this point in time no QoS provided in the Internet, its applications cannot negotiate their QoS requirements with the network nodes and install or reserve the resources needed. As a result, today's Internet applications must accept what they get – good or bad.

Several techniques have been developed which allow applications to adjust their operation to dynamically adapt their QoS requirements to changes in actual QoS provided by the network. Those techniques, generally called *adaptation*, enable multimedia applications to provide tolerable service on top of simple *best-effort* networks. Even though, *adaptive applications* significantly improve performance under moderate to high network load, they can only account for limited service degradation. Starting a communication session, such as for example an IP phone call, may be worthwhile only if a certain QoS level can be guaranteed. However, in order to achieve guaranteed quality, the network must be aware of the QoS properties and keep track of the guarantees already made.

1.2.5 Causes of Delay

The one-way, end-to-end delay seen by a data stream is the accumulated delay through the entire data flow pipeline including sender coding and packetization, network transmission, reception and decoding.

Some delays, such as coding and decoding, are of fixed duration while others are non-deterministic due to highly dynamic network or process scheduling conditions. For example, the CSMA/CD mechanisms of the Ethernet LAN link layer protocol introduces a variable delay which is deemed to be jitter.

The minimum end-to-end delay encompasses all time lags which remain constant for all transmitted units. The maximum end-to-end delay is determined by the sum of the minimum delay and the maximum jitter (compare with equation 1.2).

$$MIN(Delay) \leq Delay \leq MIN(Delay) + MAX(Jitter) \tag{1.2}$$

The transmission delay of packets (or cells) in the network results from the accumulation of the processing times in every intermediate router (or switch) between the source and the destination node and the transmission time on the physical links on this path. The transmission time on the network link is dependent on the physical medium and the link layer protocol. Well known link characteristics are, for example, ATM providing bandwidths including 155 Mbps and 622 Mbps, basic Ethernet (IEEE 802.3) offering 10 Mbps, or Fast Ethernet supporting up to 100 Mbps.

The processing time within the network nodes depends mainly on the forwarding mechanism in use. Switches or routers that process the packet in hardware require very little processing time. Such devices are usually found in the core of the network where many links are concentrated. Software based solutions depend on their implementations and on the processing engines. Such, so called, "slow path" routers require significantly more processing time (of the order of 1 ms) to forward a packet (or cell).

The processing time of the encoding and packetization at the sender, and the reception and decoding at the receiver depend mainly on the performance of the processors and the encoding and decoding algorithm. Some encoding formats (for example, ADPCM) require very little computation whereas others (for example, GSM speech encoding [C+89]) require significant processing power. Depending on the coding scheme, the time to coding the media frames might vary due to the differences in the media. These variations, however, are usually hardly recognizable.

Even though the processing time within end hosts is mainly fixed due the constant processing task. Variable latency might occur due to process scheduling irregularities and hence introduce jitter.

1.2.6 Causes of Jitter

Packet queuing in the network to compensate traffic bursts is widely recognized to be the main cause for delay variations. This section explains how queuing leads to jitter. Yet, queuing is not the only reason for delay variations. Jitter is also an artifact of process scheduling in end hosts and software based packet forwarding in network routers.

Packet Queuing

If all the packets of a flow traveling along the same path encounter the same queue lengths, they all experience the same transit delay. The end-to-end delay might be high, but the delay variance is zero. Thus, jitter is caused when consecutive packets experience different waiting times in queues.

Queues grow in a switch or router whenever the sum of the incoming data rates for one outgoing link is larger than the bandwidth of this outgoing link. In cases where bursty traffic competes for the available link bandwidth another noteworthy effect called *packet clustering* [Sch97] occurs. Bursty flows competing for bandwidth on a network link will build up queues on the router's outgoing interface. Thus several packets of the same flow might arrive while the first packet is still queuing. Packets of such packet clusters are then sent out very shortly after one another. Figure 1.2 illustrates how these packet clusters develop. At all subsequent hops these packets arrive (closely) together and the same might happen again. Thus, it is likely that clusters grow with the number of hops along a transmission path.

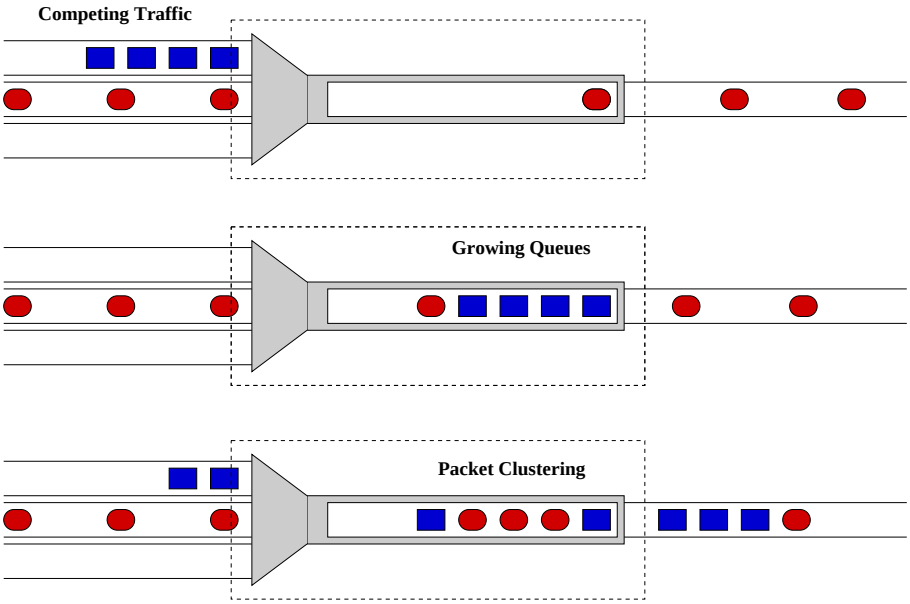


Figure 1.2: Packet Clustering

The impact of the length of the path on the end-to-end delay variation is difficult to predict. Under most circumstances the maximum delay variation increases linearly with the number of hops in the path. However, the average jitter is less dependent on the number of hops since queue waiting periods in independent queues often cancel each other partially out.

If packet scheduling is done in a strict *First-In First-Out (FIFO)* manner, a difference in the delay of two consecutive packets means that at least one queue grew or shrunk in the meantime. Under the theoretical assumption that queuing is the only cause of jitter, the maximum delay variation is bounded in the case of strict FIFO queuing. It is determined by the sum of the maximum queue lengths on the path. As more enhanced scheduling and queuing mechanisms (which support, for example, different priority queues) are used, certain flows might temporarily remain unserviced. This of course leads, in the worst case, to unbounded and unpredictable delay variations.

In order to fix the delay variation problems caused by queuing mechanisms in the network, delay sensitive applications need to deploy services which enable the total time spent in the queues to be limited. Resource reservation mechanisms such as RSVP (see section 2.3.1), for example, are capable of negotiating the maximum end-to-end delay. Nothing can be done about the transmission path length. Widely used networks do not provide shortcuts in order to limit the hop count.

Scheduling

Besides the jitter caused by queuing in network nodes, process scheduling within the end user hosts and software based switches or routers is often partially responsible for the delay variations.

Most current operating systems such as UNIX derivatives like Linux, FreeBSD, Solaris, and other operating systems such as Windows 95 or Windows NT provide multitasking, either cooperative or preemptive. Multitasking operating systems rely on a process scheduler that activates and deactivates the individual processes in an appropriate manner. This in turn usually prevents the processes from meeting the time constraints of time-critical or real-time applications.

The fact that hardware interrupts (for example, interrupts from a sound device) are intercepted and processed in kernel space[2], application processes which might have to perform one form of data processing (for example, audio applications have to encode and send the audio data) are not scheduled instantaneously. The variable time lag between the hardware interrupt and the scheduling time also causes delay variation.

The significance of jitter caused by OS scheduling depends on the scheduling granularity which determines the average and maximum waiting time until a process gets rescheduled and the maximum end-to-end delay bounds.

Experiments with our audio application showed that in the case of real-time audio streaming, jitter caused by end host scheduling can be significant. These applications demand a maximum end-to-end delay of only a few hundred milliseconds (see section 1.3.1.2). Since transmission and processing delays in packet audio already consume most of the acceptable delay (of the order of hundred milliseconds), the additional delay caused by process scheduling should be as small as possible. In addition, in the case of live audio communication, the audio data is usually sent in small packets that encompass as little as 20 ms or 40 ms audio samples. Thus, if the scheduling delay exceeds this time, the packet will be late leaving the sending host.

Running Windows 95 maximum scheduling jitter of up to 1.8 seconds has been experienced. Especially if large disk operations are performed (for example, starting memory consuming applications like Netscape's Web browser), such immense delay variations might occur. With Free-BSD and Linux (without any real-time kernel support), maximum scheduling jitter up to several hundred milliseconds has been measured. The latency could be caused by the fact that disk access is also interrupt driven; disk operations may by implemented such that they are performed within the interrupt routine in order to improve overall system performance.

Pure single task operating systems such as MS-DOS, or operating systems which allow applications to process interrupts such as Mac-OS, are generally capable of meeting the

[2]This terminology is used in the context of UNIX operating systems; Microsoft Windows systems call it "Ring 0".

strict time constraints of real-time audio applications. These operating systems do not distinguish between processing in kernel space and user space[3], and hence simplify process application code within hardware interrupts. Upon a hardware interrupt, the program counter is simply set to the user application which processes the interrupt (for example, captures the audio data from the sound device, encodes the audio samples and sends them on the network). No context switch between kernel space and user space, and between processes is required.

Operating systems that make a strict distinction between kernel and user space have three basic options to work around the scheduling problem. The first option is to modify the kernel process scheduler in order to get immediate control over the processing upon a time critical event. The second option is to use process priorities and assign high priorities to time-critical processes. The third option is to implement the time critical operations of the application in kernel space. Thus, the time-critical data processing would be performed in kernel space whereas the user space part of the application would interface the kernel module only for initialization and control purpose.

Finally, one can conclude that the only way to prevent jitter caused by irregularities in end hosts and network routers is to use real-time capable operating systems (such as Mac-OS or MS-DOS), special real-time enabled operating systems (such as RTOS, real-time Linux, etc.). The latter are topics of current research.

Jitter Compensation

Delay jitter caused by either the queuing within the network or by process scheduling increases the total end-to-end delay of transmitted packets. Since for real-time or interactive applications the end-to-end delay is probably the most crucial property, it is important to keep the delay variation as small as possible.

Packets, which miss their playback point (even by only a few milliseconds), are usually immediately discarded by the receiving application (or a filter such as a traffic shaper within the network). Packets arriving only a little late might cause the receiver to adapt its receiving buffering to accommodate these packets in future. Generally, delaying the playback point improves the playback quality (less packets are discarded) but decreases the responsiveness of real-time applications (more total end-to-end delay). This shows the trade-off between interactivity and reliability (accuracy or fidelity) that aggravates QoS based media streaming.

Adaptive playout estimation mechanisms enable optimal buffer adjustment depending on instantaneous network characteristics. See section 3.1.4 for further discussions.

[3]In Microsoft Windows this is called “Ring 3”.

1.2.7 Causes of Packet Loss

Packet switched networks are often unreliable in nature. In particular, significant parts of the Internet suffer greatly from erroneous data transmission, especially loss of packets. Packets are frequently discarded due to queue overflows in routers or end-user machines. As a result, the packet loss rate is an important QoS property for Internet multimedia application.

When packets carrying video data are lost, the video application cannot update the frames adequately. The image may become inconsistent (for example, moving objects appearing twice) or may change abruptly upon arrival of the consecutive packets. However, in audio applications, packet loss leads to crackles and gaps in the audio signal which makes speech difficult to understand and music less enjoyable. The human eye is known to act as an integrator of visual information whereas the ear acts as a differentiator. Another fact is that visual data carries in general more implicit redundancy than audio signals. Thus, in general one can conclude that packet loss within audio streaming is more disturbing for human listeners than erroneous video transmission.

Whereas some packets are lost during the transmission from the source to the destination, most "lost packets" are consciously discarded for several reasons:

First, packets are most frequently dropped because of congestion within the network. If a network node runs out of buffer space or, in other words, the packet queues overflow, packets must be discarded. A router usually has incoming buffers, system buffers and outgoing interface buffers. If packets are dropped due to an incoming queue overflow, it is called an input drop. Such input drops occur when the router cannot process packets fast enough. Packet loss due to input drops should not normally appear, since it is the result of a badly engineered system. Output drops, in contrast, occur when the outgoing link is too busy. This clearly is not a design problem of the router but an issue of available network bandwidth on the link.

Second, routers use packet dropping as a mechanism to avoid congestion in the network and prevent queues from reaching their maximum limits. One such technique is known as *Random Early Detection (RED)* mechanism (see section 2.2.2). By dropping packets before the queues hit their maximum limits, sophisticated transport protocols such as TCP (see section 2.2.2) can *early detect* potential congestion and, as a result, reduce the data rate. Even though it has been proven that this control mechanism fairly shares the network bandwidth among its users, it only works in an equitable manner as long all transport protocols play according to the rules. For example UDP (see section 2.2.1) does not back off its transmission rate when congestion occurs. Note that it is also impossible for UDP to deploy transmission control mechanisms due to the lack of feedback information. Since the UDP portion of the Internet traffic becomes larger, TCP's RED mechanism becomes less effective and rather disadvantageous for TCP traffic.

And last, damaged packets as a result of erroneous data transmission are discarded. Bit errors are usually recognized due to the checksum provided within the packet header;

these checks are often done on multiple levels (for example, the Ethernet link layer and TCP/UDP transport layer). Reliable protocols like the Ethernet link layer protocol or the TCP transport layer protocol initiate retransmission of damaged packets, whereas unreliable protocols such as UDP simply drop the packets. Packets dropped due to bit errors, however, become less common in today's fiber networks. Within wireless networks, in contrast, bit errors are frequent.

In the current Internet, packet loss is often bursty in nature. Packet clustering (see section 1.2.6) due to congestion in the network increases the likelihood of consecutive packet loss. Experiments [B+97a] confirm this theory. Another analysis performed at INRIA [BC95] shows, however, that the probability of consecutive packet loss decreases with the number of lost packets in a sequence.

The significance of packet loss depends usually on the application. In the case of non-time-critical applications, packet loss can easily be fixed by means of retransmission. However, if time constraints do not allow such additional delays (caused by retransmission), packet loss becomes a real problem. More sophisticated mechanisms such as *Forward Error Correction (FEC)* (see section 3.1.2) are developed in order to recover from packet loss. This *in-band* error correction mechanism recovers loss by means of redundancy and thus performs only well as long as lost packets are isolated to some extent. Another mechanism called *traffic shaping* (see section 3.1.1), which resolves problems of packet clusters by restoring the original (transmission) time gaps between consecutive packets again, plays an important role in conjunction with FEC mechanisms. Although the additional end-to-end delay introduced by FEC mechanisms is relatively small compared to retransmission techniques, the maximum delay constraints of real-time or interactive applications prevents FEC mechanisms to take more than a few consecutive losses into account.

Delay, Jitter and Packet Loss

Finally, the extent to which the QoS properties of delay, jitter and packet loss are correlated in the context of packet switched networks such as the Internet must be considered.

Intuitively, one expects that flows with high packet loss rates also have high jitter. While this is certainly true for heavily congested paths, flows with high packet loss rates do not necessarily have high jitter in the arriving packets. Also, flows with low packet loss rates might have high jitter.

Assuming that jitter is mainly caused by packet queuing within network nodes, as a consequence of packet bursts and clusters (see section 1.2.6), it becomes clear that there is no implicit relationship between packet loss and delay variation for low congested paths. Full queues do not lead to jitter; it is the growing and shrinking of queue length that introduces delay variations. Therefore, bursty flows competing for bandwidth on the outgoing link are responsible for delay variations independent from the degree of long-term congestion.

Even if the available bandwidth is hardly used, there might be remarkable delay variation caused by bursty traffic.

Another issue regarding jitter and packet loss arises from the maximum end-to-end delay that is tolerable for real-time or interactive applications. Packets, which exceed the tolerable end-to-end delay due to very high jitter, must be discarded. Late packet arrival is for these applications equivalent to packet loss.

1.3 Application QoS Requirements

Historically, the Internet was designed for *discrete data traffic* and *elastic applications*. For this purpose, a packet-switched architecture where data packets are treated as independent units (or *datagrams*), seems to be a very flexible and simple approach. As a result, the Internet has evolved into a *packet-switched store-and-forward* network from its beginning. Due to the standing requests of a steadily increasing number of users, the network requirements for continuous media and real-time streaming applications changes drastically.

In continuous media, especially video and audio, data has inherent temporal and spatial relationships that must be carefully respected. Violations degrade the quality of application performance drastically or even make these applications useless. The perceived quality of media streaming applications is considered to be very closely related to the QoS provided by the network. The requirements of *time-critical applications* are commonly expressed as a set of values representing bandwidth, delay, jitter and loss rate constraints for the system (or network), known as QoS parameters (see section 1.2.2).

In general, *continuous streaming applications* can cope with QoS that is significantly lower than *real-time streaming applications*. The lack of strict absolute time constraints allows buffering mechanisms (see section 3.1.4) to compensate for long end-to-end delays and retransmission techniques to resolve problems caused by high packet loss rates. *Real-time streaming applications*, on the other hand, can exploit buffering techniques only to a very limited extent, otherwise they violate their end-to-end delay constraints and, as a result, lose their responsiveness. Retransmission techniques introduce too much additional delay in current wide area networks.

The following sections investigate the QoS requirements of real-time streaming applications in detail. The QoS requirements of the different media, namely audio and video, are analyzed separately.

1.3.1 QoS Requirements for Real-Time Audio Streaming

This section examines the QoS requirements of real-time audio streaming applications. Since most applications require either voice or high quality sound encoding, these two classes are examined in particular.

1.3.1.1 Throughput

The throughput requirements of audio streaming applications depend entirely on the encoding scheme used for the audio data transmission. The encoding format is usually determined by the required sound quality of the application. Tools which simply transfer voice information usually deploy other encoding techniques – especially designed for the purpose of voice data transmission (for example, *Voice Coder (VOCODER)*) – than applications which transmit high quality music information.

A. Voice Encoding

The traditional digital voice encoding technique, known as 64 kbps *Pulse Code Modulation (PCM)*, corresponds to the sound quality everybody knows from the public telephone system. It is thus referred to as *Telephone Quality* audio. The encoding scheme is defined within the ITU G.711 standard. The mono analog signal is sampled 8000 times per second and each sample encoded in 8 bits. No compression is used. The resulting bit rate of telephone quality sound is therefore $8bits \times 8000Hz = 64kbps$.

In the 1980s a number of encoding and compression techniques were developed enabling more efficient digital voice encoding than G.711. Telephone quality can also be achieved with only 32 kbps simply by applying a more sophisticated encoding technique, known as *Differential Pulse Code Modulation (DPCM)* – a loss-free encoding. Slightly lower voice quality can be provided with *Adaptive Differential Pulse Code Modulation (ADPCM)* encoded digital voice at 40, 32, 24, and 16 kbps. More recent encoding algorithms (for example the *Linear Predictive Coding (LPC)* or *Code Excited Linear Prediction (CELP)* voice coder) can reduce bit rates as low as 2.4 or 4.8 kbps for digital voice.

Voice Quality	Encoding Technique (Standard)	Bit Rate
Telephone Quality	PCM (G.711)	64 kbps
Telephone Quality	DPCM	32 kbps
(Lower) Telephone Quality	ADPCM (G.721, G.726, G.727)	40, 32, 24, 16 kbps
Lower Telephone Quality	LD-CELP (G.728)	16 kbps
GSM Phone Quality	GSM	13 kbps
Low-bandwidth Voice	CELP (Federal-Standard-1016)	4.8 kbps
Low-bandwidth Voice	LPC-10 (Federal-Standard-1015)	2.4 kbps

Table 1.1: Voice Quality Encoding Schemes and Throughputs

B. High Quality Sound Encoding

CD Quality is commonly recognized as a high quality sound encoding. The CD audio standard is based on sampling the analog signal at 44.1 kHz, each sample being coded

with 16 bits. The result is 705.6 kbps for one monophonic channel. As compact discs are stereophonic, the throughput required to transmit a full stereophony sound in CD quality is 1411.2 kbps.

Within the last few years several encoding or compression techniques for CD quality sound have been developed (see also [Fro97, Gadml]). MPEG Layer-1 enables stereo CD quality encoding with a bit rate of 384 kbps. It should be noted that both stereo channels are multiplexed in the same stream. The MUSICAM scheme, adopted for MPEG Layer-2, allows encoding stereophonic CD quality sound with "medium" bit rates of 248 or 192 kbps. More advanced encodings (for example, MPEG Layer-3 using perceptional coding) achieve near CD quality at 64 kbps per audio channel.

Table 1.2 summarizes the throughput requirements of various audio types of audio streams.

Sound Quality	Encoding Technique (Standard)	Bit Rate
CD quality	CD-DA (stereo)	1.4 Mbps
CD quality	MPEG Layer-1 (stereo)	384 kbps
Near CD quality	MPEG Layer-2 (stereo)	192-248 kbps
Near CD quality	MPEG Layer-3 (stereo)	128 kbps
Improved CD quality	MPEG (sound studio, stereo)	768 kbps

Table 1.2: Sound Quality Encoding Schemes and Throughputs

Based on these findings one can conclude that the throughput requirements for real-time, high quality sound transmission are relatively high (although sophisticated compression mechanisms are used) compared to the throughput users experience in the public Internet. This is, in part, why current research in the area of real-time audio streaming focuses on low-bandwidth voice data.

1.3.1.2 Delay

The transit delay requirements for the transmission of continuous audio streams are highly dependent on the multimedia application. In the case of pure live audio data distribution (uni-directional transmission), long delays are usually tolerable. Large receiver buffers can be deployed to compensate for high delay variations and irregularities in the network and end systems. This of course is not the case for interactive applications such as Internet Telephony or live audio conferencing systems. Interactivity, especially human conversation, demands high responsiveness. The two-way or round-trip delay of the streaming application is crucial.

The impression of "real-time" which users experience from responsive applications is subjective. User studies for the ITU indicate that most telephony users perceive communication with round-trip delays greater than approximately 300 ms as simplex connections

rather than duplex communication. However, depending on the application and user perception, more tolerant users are often satisfied with delays of 300-800 ms [G.196]. Conversations with a round-trip delay close to a second cannot easily use "normal" social protocols for talker selection.

For duplex audio transmission, a technical difficulty lies in the echo that may be audible if the end-to-end round-trip delay exceeds a certain threshold, and no particular measure (such as the use of directional microphones and speakers, or echo canceling systems) is seen to limit the echo. The ITU has defined 24 ms as the upper limit of one-way transit delay for which echo canceling is not required.

1.3.1.3 Delay Jitter

Streaming of live audio is probably the most sensitive media type to delay variations. If packets carrying the audio information arrive with a wide distribution of transit delays, the receiving system needs to wait a sufficient time, called *buffering* or *playout delay*, before playing back the data in order to ensure that most of the delayed blocks arrive in time. Otherwise, a significant number of packets would arrive late. The gaps in the signal, caused by late and lost packets, result in audible artifacts. This results in sound quality that is intolerable.

Receiver buffering mechanisms temporarily store incoming packets in a so called *buffer* until their playout point. The packets can then be played out smoothly without gaps in the signal. Buffering mechanisms are also often referred to as *delay compensation*. Although delay compensation clearly has advantages, there are two possible drawbacks of this technique. First, an additional delay is introduced at the receiver. Second, sufficient buffer memory must be available at the receiving system.

The process of determining the best buffering or playout delay is commonly called *Playout Delay Estimation* (see section 3.1.4). It is dictated mainly by the following two parameters:

- The maximum overall delay that the application or the end user can tolerate.

  In the case of interactive audio streaming, the maximum total delay is very restrictive. Since a large portion of the delay budget is consumed by network transmission and the processing in the end systems, additional delay introduced by network jitter and scheduling irregularities in the end systems should be minimized.

- The buffering capabilities of the receiving system.

  Even though the total delay might not be the limiting factor in all cases, the available memory in the end system, especially in small or mobile end devices, restricts the buffering delay. A delay of even a few seconds of high quality audio, for example, would require a considerable buffer size.

1.3.1.4 Reliability

It is important to note that bit errors usually lead to dropped packets within Internet communication. Therefore, only packet loss needs to be considered when examining the reliability requirements of Internet media streaming applications. Bit errors are dealt with on the transport layer; user applications need not consider them.

It is commonly recognized that humans are far more sensitive to erroneous audio transmission than to defective video transfer. This is due to the different processing of audio and visual information. Thus, QoS requirements for audio with respect to error liability are very strict. The maximum error rate tolerable within audio communications is highly dependent on the application[4], the encoding scheme[5], and the sensitivity of the individual human user.

One study [Jay80] concludes that no more than 5% of erroneous audio data can be tolerated in human conversations. Another study [Sch97] discovered that a packet loss rate of 1% is clearly noticeable as a crackle. Up to 13% of packet loss of voice information still allows words to be understood, but there are many crackles in the signal. Loss rates of 20% still allow sentences to be understood. This is due to the redundancy in human language. Non-redundant information like numbers get lost. Also, speakers with a (strong) accent are very hard to understand. At 25% packet loss only parts of phrases are understandable. Higher packet loss rates make audio voice transmissions for most people totally useless.

Packet losses within real-time audio streaming cannot simply be resolved by means of retransmission, since the end-to-end delay constraints would be greatly exceeded. If only few consecutive packets are lost, techniques that replay the last frame(s) rather than playing no sound mask the problem. It should be noted that gaps in the signal are immediately recognized by the listener (except during silent periods). Other techniques suggest extrapolating the missing information by determining an approximate value from previously received frames. A similar technique interpolates missing block based on the predecessor and the successor blocks [T+96].

Both extrapolation and interpolation are called *predictive* techniques, as their approach is to provide estimates for missing information. Deploying the principle of these predictive techniques for transmission error recovery is often referred to as *error concealment*.

Summarizing, one can conclude that interactive real-time audio streaming has very strict end-to-end QoS requirements, especially with respect to the end-to-end delay, jitter and reliability. The throughput requirements are less demanding.

[4]For example, audio artefacts in high quality music are usually less tolerable than erroneous voice information.

[5]It should be noted that some encoding techniques generate packets of different priority and thus it depends which packets are lost; others add redundancy to the packets which enables recovery from most packet losses.

1.3.2 QoS Requirements for Live Video Streaming

This section introduces the QoS requirements for live video streaming applications. Since this thesis focuses mainly on the issues of packet audio, only a brief discussion of video issues is presented. The aim is to highlight the main differences between audio and video in the context of real-time media streaming.

The following four classes of video quality are examined in detail:

Broadcast Quality TV: There are currently only two standards, either NTSC, which specifies a frame rate of 30 fps and a vertical resolution of 525 lines with 858 samples per line, or PAL/SECAM, which defines 25 fps and 625 lines of vertical resolution with 864 samples per line.

VCR Quality TV: This quality is observed when recording a TV broadcast on a regular consumer VCR of VHS quality. The resulting resolution is about half PAL/SECAM broadcast-quality TV.

Video Conferencing: Low-bandwidth video conferencing operates at about 128 kbps. Two aggregated basic ISDN channels are sufficient to provide the necessary bandwidth. The H.261 compression standard [H.293] has been developed to support video telephony. This encoding scheme is particularly suitable for video sequences with little movement (for example, head and shoulder video conferencing). Moving pictures can be encoded at rates of $p\times$ 64 kbps, where p is in the range 1 to 30. The picture scanning format *Common Intermediate Format (CIF)*, defined in relation to H.261, specifies a resolution of 352 pixels per line and 288 lines per frame. To achieve data rates with less than 128 kbps, the frame rate is limited to 5-10 fps. H.263, a new standard which has recently emerged, is intended for very low bit-rates ($<$ 64 kbps). It is derived from optimizations of the H.261 and MPEG-1 coding algorithms.

Animated Images: A film of single compressed Images (usually GIF [Ger87] or JPEG [JTC93] encoded) is transmitted. The quality of the video depends on the size and colors of the single images and on the available bandwidth on the network path. This video quality class adapts the throughput requirements to the currently available end-to-end bandwidth. It varies from a frame rate of 0 to the maximum frame rate supported. The throughput requirements can be further reduced by sending only the differences between subsequent images rather than the whole picture [Ger87].

1.3.2.1 Throughput

The throughput required for real-time, uncompressed broadcast quality TV results from the number of samples per line given in the definition, which corresponds to the samples for the luminance, plus 360 samples, which are required for the color difference irrespective of

the original analog signal. The number of active lines per frame is lower than the number of lines given in the definition. Only 484 active lines are used in NTSC-compatible mode; 576 lines in PAL/SECAM-compatible mode.

Compressed broadcast quality TV requires as little as 6 Mbps. Existing implementations of the MPEG-2 compression standard operate at this rate. It is expected to reduce the bit rate to 2-3 Mbps (4 Mbps) for quality equivalent to that of NTSC (PAL/SECAM) broadcast. Compression schemes such as MPEG-1 or DVI (Digital Video Interactive) provide off-line compression to 1.2 Mbps for quality similar to VCR quality. The bit rate of 128 kbps, required for CIF encoded video conferencing quality, is specifically designed for low-bandwidth links. Work is underway by the MPEG group to define schemes that can provide video conferencing quality with as little as 32 kbps or even 4.8 kbps within the new MPEG-4 standard.

The throughput requirement for animated images depends on the image size, the image encoding, and the rate at which the pictures are captured. Video systems, like the WebVideo tool [FW97] developed at University of Ulm (Germany), adapt the frame rate depending on the currently available bandwidth along the network path. Common data rates experienced in today's public Internet are in the range of 0-64 kbps.

Table 1.3 summarizes the throughput requirements of various types of compressed digital video.

Video Quality	Encoding Technique (Standard)	Bit Rate
Broadcast Quality TV	MPEG-2	3-6 Mbps
VCR Quality TV	MPEG-1, DVI	1.2 Mbps
Video Conferencing	H.261 (CIF)	128 kbps
	H.263	$<$ 64 kbps
	MPEG-4	32, 4.8 kbps
Animated Images	JPEG, GIFF, DIFF-GIF	0-64 kbps

Table 1.3: Video Quality Encoding Schemes and Throughputs

From this brief overview of different quality video encodings, it is clear that the throughput requirements of video streaming are significantly higher than the ones of audio streaming. Furthermore, a comparison of these throughput requirements with the throughputs that are experienced in the Internet today clearly shows that high quality video cannot be transmitted on the public network. Low-bandwidth video of conferencing quality is already hard to manage, since end-to-end bandwidth of 128 kbps for a single application already requires good end-to-end connectivity. Modem connections are not sufficient. Thus, low-bandwidth video encoding based on animated images with flexible bandwidth requirements is currently favorable for Internet real-time video streaming.

1.3.2.2 Delay

The delay requirements of video streams depend on whether the video stream is transmitted simultaneously with an audio stream for synchronous presentation, or not. In the case of synchronous playback of both media types, the requirements on the transit delay and the jitter are usually dictated by the audio. High quality video and audio like broadcast or VCR quality TV demands coarse synchronization such as "lip-synchronization". Thus, the delay variation between the audio and video playout should be less then 50-100 ms [T+96].

Low-bandwidth quality video with only a few frames per seconds requires only rough synchronization if audio is available. The delay variation between the audio and video should then be less than about 400 ms [T+96].

If only video is presented (without audio), the delay depends entirely on the application. If the application is interactive and response time is important, the delay, of course, should be as little as possible. The delay demands of interactive, real-time video are similar to the ones of audio.

Video playback applications without interaction such as VCR type video playback tools have only very little demands on the delay. Several seconds are usually easily tolerable.

1.3.2.3 Delay Jitter

As long as the video and audio is synchronized, the jitter requirements of the video transmission are dictated by the audio. Otherwise, small or moderate delay variations are still tolerable. While in the case of audio streaming small delay variations result immediately in spurious sound quality (if no delay compensation is deployed), varying playout delays of video frames are less disturbing. This is due to the fact that human sound recognition is more sensitive to irregularities in the signal than the eye.

The amount of tolerable jitter mainly depends on the video quality and in particular the frame rate. In the case of high quality video with frame rates of 25-30 fps, jitter above 50 ms will be recognized in most cases. On the other hand, if low-bandwidth video quality with 5-10 fps is used, jitter of about 100 ms will hardly disturb the user.

If the jitter experienced by video packets exceeds the tolerable limit, delay compensation mechanisms, as described in the case of audio, must be deployed. If synchronization is required, the playout delay estimations of the audio and video must be adjusted. The playout point of the video should not vary from the audio by more than 50-100 ms.

1.3.2.4 Reliability

As mentioned earlier, humans are less sensitive to erroneous video transmission than to defective audio transfer. The reason is simply the different processing of audio and visual

information. Therefore, the QoS requirements for video with respect to error liability are less strict than for audio.

The maximum error rate tolerable within video streaming is highly dependent on the application. Missing frames usually result in jerky movement. The degree of disturbance depends on the video quality level and especially the frame rate. Motion interruption in high quality video is immediately recognized, whereas in low-bandwidth video a missing frame might not be noticed.

Unlike the case of audio playback, a missing frame does not lead to a gap in the signal. The user still perceives an image, even if it is an old image. It is only the motion which is intermittent, whereas in the case of audio playback the signal is completely missing for a period of time. As in the case of audio transmission, predictive error concealment mechanisms such as extrapolation and interpolation are often deployed in order to reduce the problem of frame losses.

Since the human eye acts as an integrator of visual information rather than as a differentiator like the ear, gaps in the signal are not as noticeable. Thus, erroneous video transmission and in particular packet loss is more tolerable than defective audio transfer.

Summarizing can be stated that video streaming requires significantly more available bandwidth than audio, whereas the end-to-end QoS requirements with respect to jitter and reliability are less strict and more scalable than for audio. However, if the video signal is to be synchronized with the audio, the stronger requirements of audio streaming usually dictate the transmission characteristics of the video.

1.4 Summary

The introductory chapter provides an overview of related work, with respect to real-time audio streaming applications, which are currently carried out within Internet research. Further related work is presented throughout the thesis where appropriate.

A general classification of Internet traffic is presented, placing real-time streaming applications and media traffic within the context of general Internet applications and traffic characteristics. Moreover, the specific concerns and problems of real-time streaming are discussed in more detail.

Since the thesis emphasizes the QoS issues of real-time audio streaming, an in-depth introduction of the QoS terminology is presented. The QoS parameters that play an important role within Internet communication are: delay, jitter, throughput and reliability. The work focuses especially on real-time streaming in the Internet. As a result, a discussion about the QoS experienced in the public Internet is included. The results can be summarized as follows:

- Today's Internet communication is based on the simple *best-effort* service model, and thus, no explicit support for QoS is available yet.
- The end-to-end delay of a data stream is the accumulated delay through the entire data flow pipeline including sender coding and packetization, network transmission, reception and decoding.
- Network delay in the Internet is mainly a result of queuing and processing within network routers. Link transmission delays are usually small.
- Jitter is caused either by the dynamic changes of queues in the network or by process scheduling irregularities in end hosts and network routers.
- Unreliability in Internet communication is mainly expressed by the packet loss rate. Packet loss is a result of routers discarding packets because of temporary congestion or as a precaution in order to avoid congestion.

In order to provide a foundation for further discussions on QoS issues of real-time streaming, the QoS requirements of media streaming applications, are presented. The important results can be outlined as follows:

- The throughput requirements of video streaming are significantly higher than for audio streaming.
- Humans are far more sensitive to erroneous audio transmission than to defective video transfer.
- Interactive audio streaming demands very small end-to-end delays. Users usually perceive communication with round-trip delays greater than 300 ms as simplex connections.
- Live audio is highly sensitive to delay variations and hence demands delay compensation at the receiver.

Conclusions:

1. Interactive audio streaming has very strict end-to-end QoS requirements, especially with respect to the end-to-end delay, jitter and reliability. The throughput requirements are less demanding.
2. Video streaming requires significantly more bandwidth than audio. The end-to-end QoS requirements with respect to jitter and reliability are less strict than for audio. However, if the video signal needs to be synchronized with the audio, the stronger requirements of audio streaming apply.

In the next chapter several protocols that play an important role in Internet multimedia streaming are discussed. The structure is inspired by the OSI reference model [Tan96]. The chapter, therefore, groups and discusses the protocols following the OSI layering. Chapter 3 introduces and discusses different approaches to improve the QoS for real-time media streaming applications. It distinguishes between application layer techniques and mechanisms that are provided on the network layer. In chapter 4 WebAudio, the real-time audio streaming application which was implemented in the context of this thesis, is introduced. The application architecture and implementation issues are discussed in detail. Chapter 5 presents the results of various experiments accomplished with WebAudio. The experiments can be grouped into those examining the proper operation of the application in different networks environments, those exploring the resource reservation capabilities of WebAudio, and those investigating the efficiency of various packet classification approaches. Finally, chapter 6 summarizes and concludes the work. Further development work on WebAudio and future research with respect to RSVP is described at the end of this chapter.

Chapter 2

Internet Multimedia Protocols

This section introduces most of the Internet communication protocols used by state-of-the-art multimedia streaming applications. The protocols used in the real-time audio streaming application developed in the context of this thesis are described in full detail.

An overview of current protocols is shown in Figure 2.1. It associates the individual protocols with their OSI layers. Unfortunately in many cases it is hard to classify streaming protocols according to the OSI reference model. Many modern protocols have a rather "vertical" design; or in other words, they cross the boundaries of one layer. An example is RSVP (see section 2.3.1) which provides apart from the network level resource reservation control also an application level interface. Applications initiate and control the reservation mechanism via the application level interface known as *RSVP Application Protocol Interface (RAPI)*.

Assigning upper-layer protocols (for example, HTTP, RTSP, etc.) to their OSI reference model is even more difficult. Hence, the three top layers of the OSI model are merged into one single layer here called *Application Support Layer Protocols*. This includes all protocols above the *Transport Layer* which provide any kind of service to end-user applications.

Although it is difficult to place some protocols in the OSI reference model, they are described and examined here according to the OSI layer model.

2.1 Network Layer Protocols

The main network level protocol used within today's Internet is still IPv4, even though the next generation Internet protocol IPv6 has already been specified in 1995 [DH95]. Since IPv4 specification in 1981 [Pos81], IPv4 has undoubtedly evolved to be the most widely deployed network protocol ever.

The Internet protocol is designed for use in interconnected systems of packet-switched data communication networks. Its function or purpose is to move datagrams through an

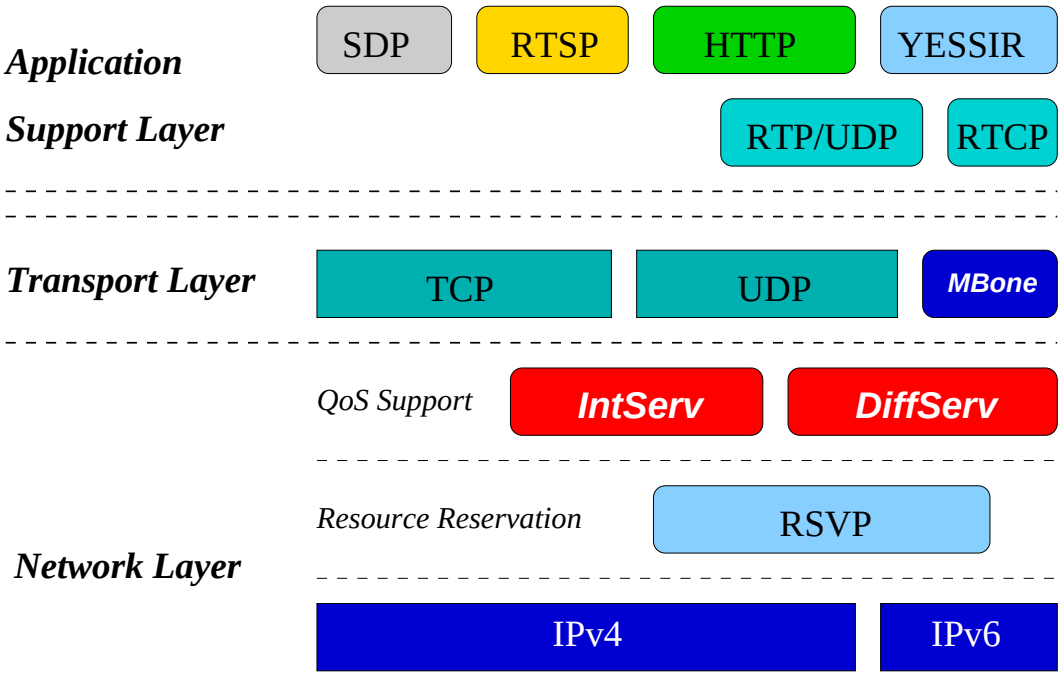


Figure 2.1: Internet Multimedia Protocol Stack (related protocols or protocols with similar functionalities have the same shading)

interconnected set of networks. This is done by passing the datagrams from one Internet module (network element) to another until the destination is reached. The selection of the transmission path and the subsequent forwarding of datagrams along this path is called *routing*. The datagrams are routed from one Internet module to another based on the interpretation of the Internet address in the datagram. According to the Internet communication model, datagrams (or packets) are treated as independent entities and, as far as the network subsystem is concerned, are unrelated to each other. The Internet does not support connections or logical or virtual circuits. End-to-end connections have to be emulated at a higher layer (for example, the transport layer).

IPv4 serves as the network layer protocol for the well known *Transport Control Protocol (TCP)* (see section 2.2.2) and *User Datagram Protocol (UDP)* (see section 2.2.1) that are used within all of today's Internet application (for example, HTTP, Email, FTP, Telnet, etc.)

2.1.1 Internet Protocol version 6 (IPv6)

The next generation *Internet Protocol version 6 (IPv6)*, specified in RFC 1883 [DH95], was designed as a direct successor to IPv4 [1].

The concern about running short of Internet addresses was the main initial drive to develop a next generation Internet protocol with a much larger address space. Based on the exponential growth of the Internet in recent years, experts predicted that the Internet runs out of IP addresses within the next 20 years [Hui97]. Since every network element requires an unique IP address, the maximum number of elements is limited to about 4 billion. This upper bound results from the IPv4 address length of 32 bit which stretches an address space of 2^{32} addresses. Even though this number is sufficient to provide an unique IP address to about 2/3 of the world's population, the fact that the address space is hierarchically partitioned[2] in groups, accounts for inefficient allocation. Practical observations in the Internet have shown that the address allocation efficiency is less than 30% [Hui97].

The IPv6 designers made arrangements to smoothly migrate from IPv4 to IPv6. They designed the new protocol such that both versions can coexist simultaneously. The *Version* field (see Figure 2.2) allows network elements to quickly identify the Internet protocol of a packet.

Since the version 1 specification of IPv6 [DH95] was released, the IPv6 header format has already undergone several changes. According to the latest draft release of the IPWG [DH98], the IPv6 header is defined as presented in Figure 2.2.

The important changes from IPv4 to IPv6 fall primarily into the following categories: expanded addressing, header format simplification and an efficient extension (options) header mechanism, multicast and anycast capabilities to support new styles of communication, and new security capabilities.

2.1.1.1 Expanded Addressing Scheme

As shown in Figure 2.2, IPv6 increases the IP address size from 32 bits to 128 bits[3] to augment the levels of addressing hierarchy and the overall number of addressable network nodes.

The 128-bit IPv6 addresses are written as eight 16-bit integers separated by colons. Each integer is represented by four hexadecimal digits. A set of consecutive null 16-bit numbers

[1] The jump from version 4 to version 6 is rooted in the fact that version 5 had been allocated to ST, an experimental "stream" protocol designed to carry real-time services in parallel with IP.

[2] Hierarchical addressing is required to maintain efficient routing tables and achieve fast routing within the network.

[3] This address space, for example, could provide sufficient addresses to address every single byte of memory currently available on earth (without reference).

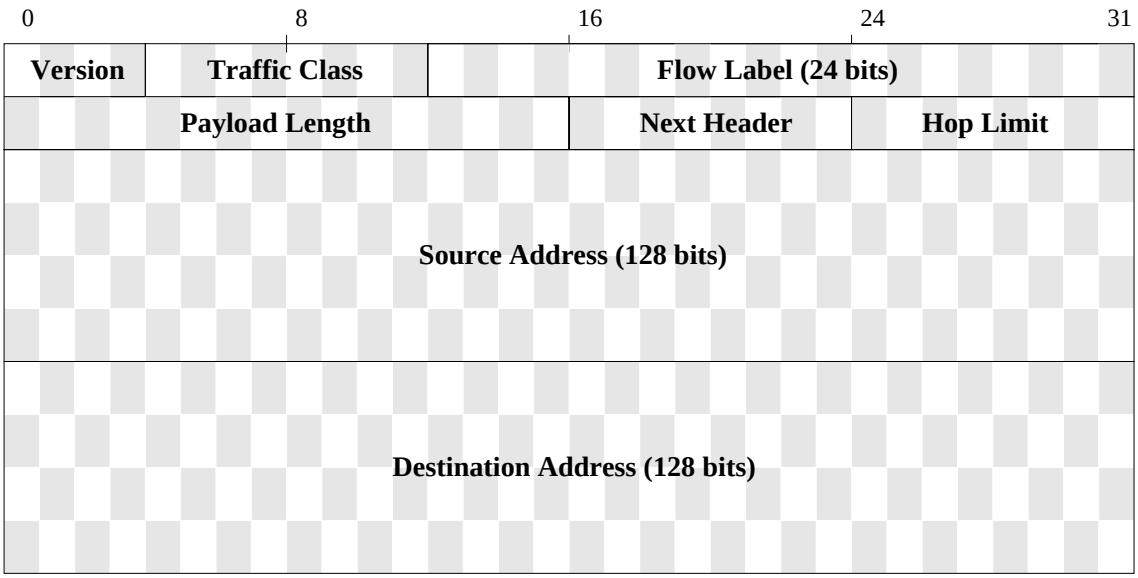


Figure 2.2: The IPv6 Protocol Header

can simply be replaced by two colons. The last 32 bits can be presented in dot decimal form.

The following example shows three typical forms of textual representations of the same IPv6 unicast address:

1080:0:0:0:8:800:200C:417A
1080::8:800:200C:417A
1080::8:800:32.12:65.10

The address size and structure enables also automatic address configurations for network nodes. A node's network address (64 bits) concatenated with the device's unique MAC address (converted to a standard 64 bit EUI-64 address) is a simple mechanism to create a unique IP address.

IPv6 addresses are assigned to individual network interfaces rather than to a corresponding node. Single interfaces may have assigned multiple IPv6 addresses of any address type, namely unicast, anycast or multicast.

The huge address space is also used to improve multicast routing by adding a *Scope* field to multicast addresses (see section 2.1.1.3). Furthermore, a new address type, called *anycast address*, has been introduced in IPv6 (see also section 2.1.1.3). On the other hand, IPv6 does not provide broadcast addresses anymore. Broadcast is simply superseded by means of multicast.

2.1.1.2 Header Format Simplification and Header Extensions

According to Figure 2.2, the IPv6 header compromises only eight header fields. Some IPv4 header fields, such as the *Header Length*, *Flags*, *Fragment Offset*, *Header Checksum*, *Options*, and *Identification*, have been dropped or made optional to reduce the common-case processing cost of packets and to limit the overhead of the IPv6 header.

Instead of using the IPv4 *Option* field [Pos81], IPv6 encodes optional network-layer information in special headers, called *Extension Headers*. They are placed in between the IPv6 header and the transport-layer header. A small number of such extension headers is already defined, each of which is identified by a distinct *Next Header* value. IPv6 packets may carry zero, one, or more extension headers. The next header field points to the next extension header in the chain. See Figure 2.3 as an illustration.

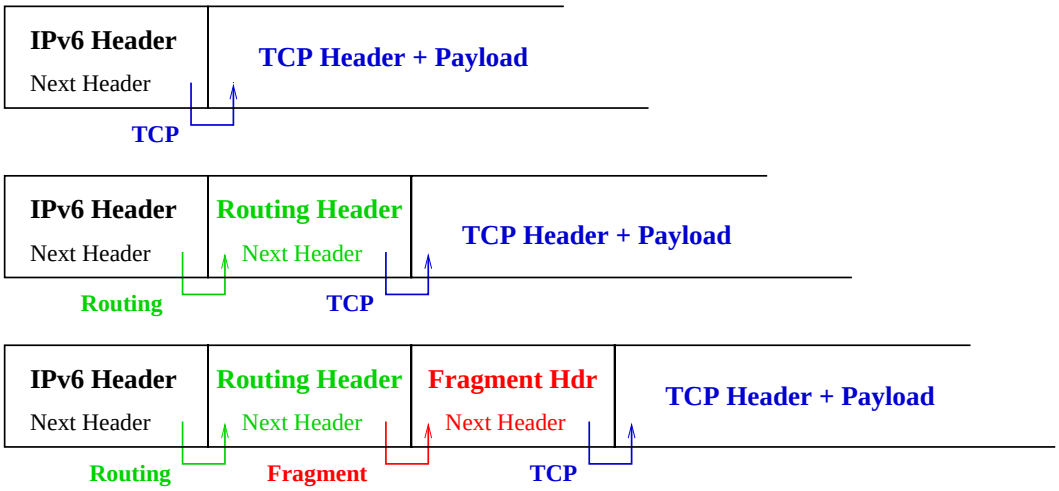


Figure 2.3: The IPv6 Extension Header Mechanism

The extension headers must have a size equal to an integer multiple of 8 octets. This guarantees that all subsequent extension headers retain a 8-octet alignment, and thus simplifies and speeds up the header processing.

If more than one extension header is used in the same packet, they should appear in the order shown here:

IPv6 header
Hop-by-Hop Options header
Destination Options header[4]
Routing header

[4]Options to be processed by the first destination of the Destination Address field plus subsequent destinations listed in the Routing header.

Fragment header
Authentication header
Encapsulating Security Payload header
Destination Options header[5]
Upper-layer header (i.e. TCP header, UDP header)

With the exception of the *Destination Options* header, each extension header should occur at most once.

In order to enable efficient packet forwarding, extension headers must not be examined or processed by any node along a packet's delivery path, except when the packet reaches its destination. There is, however, one exception. This is the *Hop-by-Hop Option* header that carries information that must be processed by every network node passed by the packet. Hence, if this extension header is used, it must immediately follow the IPv6 header to prevent nodes from having to search for this information.

Furthermore, extension headers must be processed strictly in the order they appear in the packet, since the contents and semantics of each extension header determines whether or not the following header must be examined.

2.1.1.3 Anycast and Multicast

Anycast is a new feature[6] of IPv6. The principle of anycast is very simple. Sending a packet to an anycast address delivers the packet to any, but only one, of a group of network interfaces. IPv6 anycast addresses have the property that a packet sent to such an address is routed to the "nearest" interface associated with that address, according to the routing protocol's measure of distance.

Moreover, the designers of IPv6 took advantage of the deployment of a new protocol to make sure that multicast is available on all IPv6 capable network nodes. In IPv4 multicast (sending packets to all members of the multicast group) was achieved by means of a virtual network on top of UDP/IP which is known as the *MBone* [Eri93]. MBone routers simply duplicate data packets if they have to forward them on different output ports; otherwise only one packet is forwarded. By defining an address format for IPv6 multicast addresses and defining multicast routing as mandatory, they force multicast to be a native communication mode in IPv6.

IPv6 multicast addresses can be interpreted as identifiers for a group of nodes. Nodes may simultaneously belong to a number of multicast groups. The format of IPv6 multicast addresses is shown in Figure 2.4.

[5]For options that must only be processed by the final destination of the packet.
[6]It was still a research project when the IPv6 specifications were written.

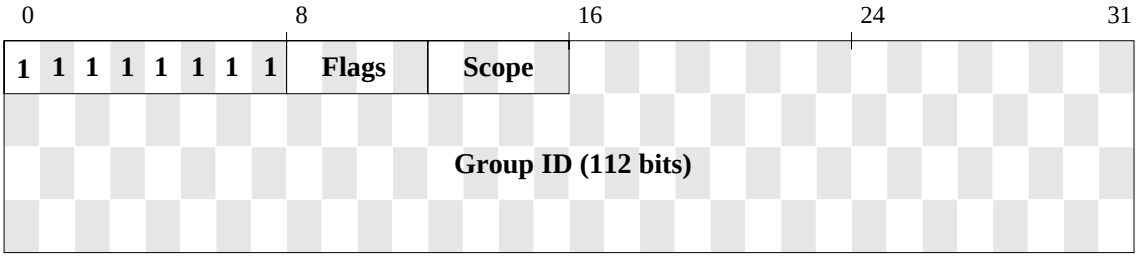


Figure 2.4: The IPv6 Multicast Address Format

The *Flags* bit set is currently only used to indicate "well-known" (permanently-assigned) multicast addresses or "transient" (non-permanently assigned) addresses. The *Scope* field is used to limit the scope of the multicast group. Currently defined scopes are node-local, link-local, site-local, organization-local, and global. Special multicast addresses are assigned to simulate link-local broadcast (all local node addresses) and router broadcast (all local router addresses).

2.1.1.4 Security Capabilities

In recent times security has become an important issue in the Internet. Today most organizations secure their data and privacy simply by barricading their networks behind firewalls. This shows that there is a great demand for security in the Internet. As a result, the designers of IPv6 have incorporated security features known as IP-level authentication and encryption.

Supporting security as native mode on the network level is a transparent means to add secure communication to all applications. The inclusion of security mechanisms as an integral part of IPv6 promotes fast integration and distribution of security within the Internet.

A more detailed description of the Internet security architecture is provided in [Atk95c]. The use of the extension headers for authentication and encryption is specified in [Atk95a, Atk95b].

2.1.2 Enhancements for Live Media Streaming

2.1.2.1 IPv6 and Performance

The IPWG has heavily learned from past experiences with Internet multimedia applications in IPv4. They avoided several of the performance bottlenecks which existed in the design of IPv4. The main improvements are discussed here.

The IPv4 *Header Checksum* field has been dropped. Since some IPv4 header, such as for example the *Time To Live (TTL)*, change at each intermediate network node, the header checksum must be verified and recomputed at each node. Eliminating this control mechanism is definitely a wise choice, especially, since current transport protocols, namely UDP and TCP, have their own checksum validation mechanisms. Dropping the header checksum of the network protocol reduces the per-packet processing time in intermediate routers. Considering that packets pass several hops until they reach the destination, this simplification may have a significant impact on the overall end-to-end transmission delay. However, in the case of data transmission in unreliable networks, such as wireless networks, the lack of network level bit error detection might cause unnecessary delivery of erroneous packets.

Removing the *Option* field of the IPv4 header and introducing the IPv6 extension header mechanism instead, also has an impact on the average packet processing time. In IPv4 all packets with a *Header Length* greater than five 32 bit words, contain options that need to be processed in a special manner by intermediate nodes. Clearly an identifier that tells whether the options have to be processed by each node or only at the destination is missing. The IPv6 header, on the other hand, is of fixed size. It definitely simplifies and speeds up the header processing. The "ordered" extension header list of IPv6 forces routers only to check the "options" which are destined for intermediate nodes. Extension headers which are intended for the end nodes can be completely ignored.

Preventing IPv6 from fragmenting packets under normal operation not only simplifies the IPv6 header — the IPv4 header fields *Identification*, *Flags*, and *Fragment Offset* become obsolete — but also improves the packet processing performance in network routers. Fragmentation not only increases the overhead associated with the fragmented packet due to the replication of the header in each fragment, but also demands additional processing at the end nodes for fragmentation and reassembly. IPv6 uses the *Path MTU Discovery* [MD90] mechanism to find the *Maximum Transmission Unit (MTU)* of a link before starting the transmission. Knowing the path MTU allows the IPv6 implementations of the end systems to perform necessary packet fragmentation. The processing burden of packet fragmentation is therefore shifted from the network routers, where high load might result in congestion, to the end systems. Moreover, if applications know the path MTU, they can avoid unnecessary fragmentation by applying proper application level framing. If packet sizes larger than the path MTU are absolutely required, IPv6 provides the *Fragment* header to indicate fragmentation and ensure proper reassembly at the receiver. However, fragmentation is strongly discouraged in cases where applications are able to adjust their packets to fit the path MTU.

2.1.2.2 IPv6 and QoS

Although IPv6 enhances the current Internet protocol with respect to QoS support and real-time media streaming, the lack of QoS support was never a reason that pushed the

development of a new Internet protocol. The growing demand for QoS support in the Internet led the IPv6 developers to introduced the following features to facilitate network QoS control.

Traffic Class

According to the version 1 of the IPv6 specification, the 8 bit *Type of Service (ToS)* field of IPv4 was simply transformed into a 4 bit *Priority* field. It was intended to enable IPv6 applications to mark the delivery priority of packets relative to other packets of the same source.

However, three reasons led the IPng working group to revise the *Priority* header. First, it is difficult to arrange fair or useful assignments of packet priorities. Note, any application performs better when using a higher priority class for its traffic. Second, it has been demonstrated that relative priorities can cause open loop transmission problems [Hui97]. Third, more control bits to improve the *congestion avoidance algorithm* of RED were needed (see section 2.2.2). As a result, the *Priority* field was transformed into an 8 bit *Traffic Class* field. The additional 4 bits were gained by reducing the *Flow Label* field to 20 bits [DH98].

The IPv6 *Traffic Class* field is used by intermediate network nodes to identify the different traffic or priority classes of passing packets.

It is still not entirely clear how this field will be used in practice. The most promising approach suggests to use the bits as in IPv4 ToS and/or Service Marking (see section 3.2.2) to provide various forms of "differentiated service" for IP packets, as an alternative to establishing explicit flows. Current experiments with IPv4 and DiffServ (see section 3.2.3) will hopefully lead to agreement on which types of traffic classification are most useful for IP packets.

Flow Label

When the designers of IPv6 included the *Flow Label* header field, they were convinced that a flow identifier was a promising extension with respect to network QoS although its usage was only vaguely determined.

According to the IPv6 specification [DH95], the flow label field might be used by the source to label packets that require special handling by intervening IPv6 routers, such as non-default QoS or real-time service. In order to classify packets belonging to the same flow, they are labeled with the same pre-defined flow label value. Therefore, the definition of a flow comes implicitly from the definition of the flow label itself. A *Flow* is defined as a sequence of packets sent from a particular source to the same (unicast or multicast) destination.

The inclusion of the *Flow Label* field provides the IP protocol with the concept of a *Flow*. As a result, network elements are now capable of classifying packets based on IP semantics alone. This allows efficient mapping of packet to their flows and hence to their flow specific processing policy (for example, QoS requirements or Class of Service).

Flow labels are assigned to flows by the source or sending nodes in an unique manner. Note, a source can never have more than one flow with the same flow label at a given time[7]. New flow labels must be chosen (pseudo-) randomly and uniformly.

The flow label properties are ideal for proper and efficient packet classification. In a recent publication at the Spie East '98 conference [S+98a], we present a detailed discussion of the impact of the IPv6 flow label on packet classification, and in particular classification within the IntServ architecture (see section 3.2.5) and RSVP (see section 2.3.1). Further discussion of this findings is presented in section 5.3.

2.1.2.3 IPv6 and Multicast

Multicasting is an excellent mechanism for data distribution, especially for group communication. It can generally reduce network utilization and the processing load on the sender node. As a result, many of today's multimedia streaming applications use multicast to distribute their media streams to all members in the group. Several multimedia applications have been designed to run on the MBone and became commonly known as the *MBone Tools* [8].

In IPv4 networks multicast communication is, however, only offered by the virtual network called MBone [Eri93]. Since IPv4 multicast implementation have to uses tunnels between the individual multicast routers, multiple copies of encapsulated multicast packets are transmitted on the same links. The transmission of multiple replicas of multicast packets on the same link results from the fact that often several tunnels are configured over a single physical link.

IPv6, in contrast, supports multicast as a native communication mode, and hence, does not rely on *MBone*-like tunnels. It is precisely by avoiding the use of tunnels for multicasting that IPv6 improves the performance of multicasting. Yet, this improvement will only be noticeable when a significant number of Internet routers support IPv6. Until then tunneling will remain.

The great advantage of having multicast fully integrated in the Internet protocol is that over time the multicast network successively increases by the number of IPv6 network nodes. Thus, eventually multicasting becomes a standard feature of the Internet (rather than being treated as a optional add-on as it is today).

[7] A discussion on how to ensure unique local flow labels is presented in [DH98].

[8] Examples are vat [JMat], vic [MJ95], rat [H+95], NV (Network Video), WB (White Board), etc.

2.1.3 Summary

The main differences between IPv4 and IPv6 are summarized in Table 2.1.

Criterion	IPv4	IPv6
Address Size	32 bits	128 bits
Header Size	20-60 Bytes	40 Bytes (fixed)
Options	0-40 Bytes header field	extension header mechanism
Checksum	+	-
Multicast	virtual (MBone)	native
Anycast	-	+
Flow Support (Flow Label)	-	+
Fragmentation	by default	on demand
Security	encapsulation	native

Table 2.1: IPv4 vs. IPv6: What are the differences?

Summarizing from this section, one can conclude that IPv6 mainly contributes to the Internet protocol in 4 directions: first, it fixes the address problem; second, IPv6 adopts simpler and more efficient versions of IPv4 mechanisms which degraded the packet processing performance in network routers (i.e. fragmentation, checksum, header size); third, it integrates successful IPv4 add-ons (i.e. multicast, security) as native parts to the protocol; and fourth, it introduces the new concepts of anycast and flows, to the Internet protocol.

Finally, three noteworthy issues regarding the extensions of IPv6 are pointed out here:

First, although the flexibility and extensibility of the IPv6 extension header mechanisms is highly acknowledged, it arises a problem if network routers rely on information of the IP payload (for example, the transport protocol port). If multiple extension headers are in use, the processing cost of skipping the headers to get to the payload increases significantly (it means, searching a linked-list) [9]. One could argue that IP routers are supposed to operate on the network level only, and hence should not rely on payload content. Nonetheless, mechanisms like packet classification (as currently deployed within IntServ/RSVP), for example, rely on transport layer information.

Second, even though IPv6 introduces the concept of a *Flow*, it does no fully make use of the concept for QoS support within multimedia communication. QoS support and in particular resource reservation mechanisms are closely linked to the concept of a "connection". Therefore, the ability of using the IPv6 flow label to "pin" an end-to-end "connection" (i.e. the route) for a data flow would be advantageous. However, since packets in IPv6 are routed only according to their destination address (or routing header extension), packets

[9] Note, the IPv6 header does not include a header size field anymore.

of the same flow may be delivered via a different path. The lack of true flow routing (or switching) capabilities has the danger that applications could lose their QoS guarantees due to a route change.

Third, the fact that IPv4 and IPv6 have completely different headers formats makes them unable to inter-operate. Therefore, during the transition, each IPv6 router requires a dual stack architecture (IPv4 stack must also be provided). In addition, IPv6 traffic that needs to traverse a network segment only capable of handling IPv4 has to be tunneled. This, of course, results in bad performance due to the processing overhead and the increased network load caused by packet replicas.

2.2 Transport Layer Protocols

2.2.1 User Datagram Protocol

The *User Datagram Protocol (UDP)*, defined in 1980 [Pos80a], implements a datagram based mode of packet-switched communication for the Internet. The transport protocol assumes IP to be the underlying network protocol.

UDP offers to applications a simple mechanism to transmit messages between processes with a minimum protocol. It is known to be a "transaction-oriented" protocol without guarantee for packet delivery, protection against duplication and promises for in-order delivery. A checksum mechanism is deployed to identify bit errors. Thus, it guarantees service with bit-error free packet transmission.

Since UDP is one of the simplest transport protocol one can think of, its protocol header is as simple as shown in Figure 2.5.

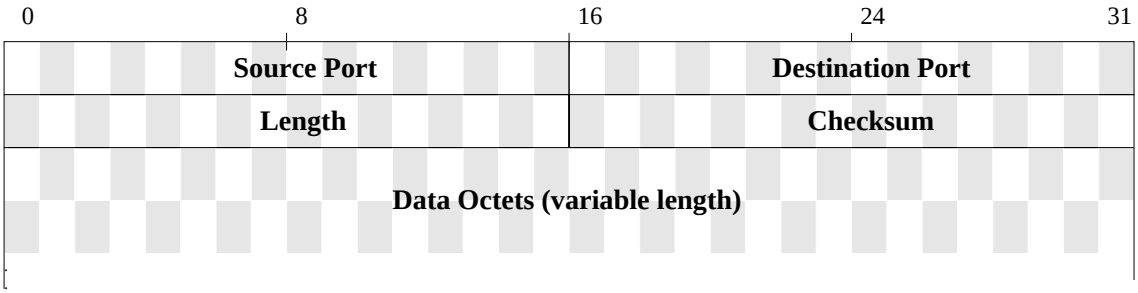


Figure 2.5: The UDP Protocol Header

To allow multiple processes on the same host simultaneous use of UDP-based communication, UDP provides a set of addresses per host, called *Ports*. Ports are defined access points for data communication. The concatenation of ports and the network and host addresses

of the IP layer, compose a socket. The binding of ports to processes is handled independently by each host. However, experiences show that it is useful to attach frequently used processes (for example, WWW, FTP, Telnet) to fixed sockets. These services can then be accessed through their "well-known" ports.

The *Source* and *Destination Port* of the UDP header specify the socket ports of the end-user processes sending and receiving the packets. The destination port is a means to demultiplex multiple data streams at the receiver. The source port is not required; if not specified, a value of zero is inserted. The *Checksum* is the 1's complement of the 1's complement sum of the payload data, the UDP header and a pseudo IP header (see [Pos80a] for further details). The *Length* simply specifies the total number of octets in the user datagram.

2.2.2 Transport Control Protocol

The *Transmission Control Protocol (TCP)* [Pos80c] is known as a reliable host-to-host protocol between hosts in IP networks, such as the Internet. It offers connection-oriented, end-to-end, inter-process communication between any pair of hosts connected to the Internet. TCP makes only very few assumption on the network protocol. It provides reliability, even on top of the unreliable datagram services offered by IP. Interfacing both, the application process and the lower level network protocol, makes TCP a general "inter-process" communication protocol for multi-network environments. Its main design features are examined and discussed here:

Data Transfer: TCP is intended to transfer streams of data octets in a bi-directional manner between pairs of hosts. The data stream is transmitted in segments (or packets) including a variable number of data octets. In TCP streaming mode, the TCP stacks on both hosts decide (see TCP flow control) when to block and forward data at their own convenience. TCP can also be operated in *record mode*. This mode, however, is hardly used anymore.

Reliability: Since TCP guarantees reliable data transmission, it must recover from data which is damaged, lost, duplicated, or delivered out of order by the underlying network protocol. This is achieved by assigning a sequence number to every octet transmitted. The *Sequence Number* field of the TCP header (see Figure 2.6) identifies the first data octet in the packet. Sending hosts wait for a positive acknowledgment (ACK) from the receiver after a certain amount of data octets is sent. If the ACK, indicating the successful receipt of data up to the *Acknowledgment Number*, is not received within a timeout interval, the sender starts retransmission. The sequence numbers are used at the receiver to correctly order segments which may be received out-of-order and to eliminate duplicates. Bit-errors in the payload data are detected by means of the *Checksum* transmit as part of the protocol header.

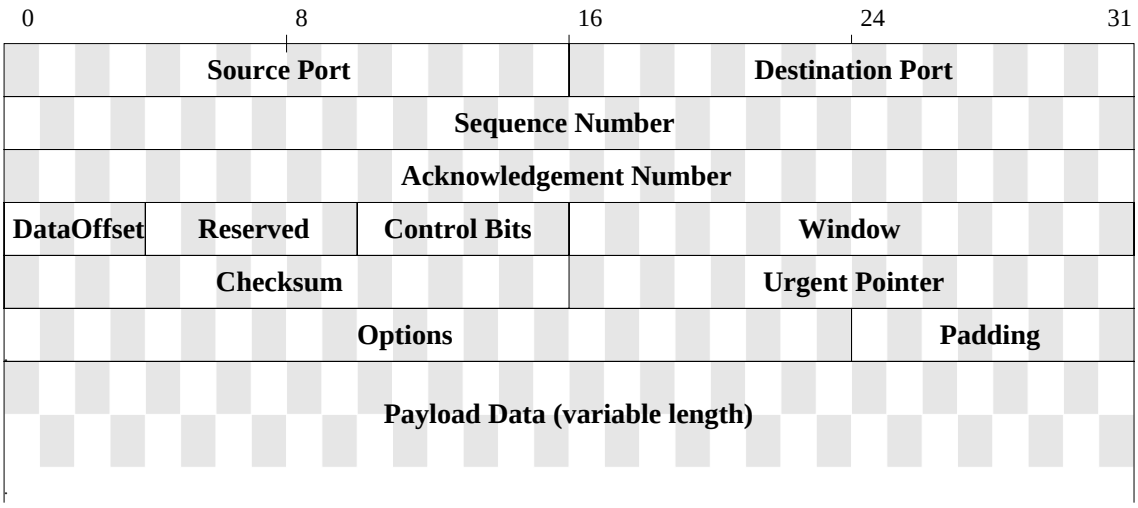


Figure 2.6: The TCP Protocol Header

Flow Control: TCP uses the well known *sliding window algorithm* as a flow control mechanism. This is a means for the receiver to control the data rate transmitted by the sender. TCP receivers simply return a *"window"* with every ACK indicating a range of acceptable sequence numbers beyond the last segment (or packet) successfully received. The *sliding window* indicates an allowed number of data bytes that the sender may transmit before receiving an acknowledgment for the first data segments. After receipt of new ACK, the sender scrolls forward its transmission window, and starts transmitting new data segments. If the sender does not receive an ACK within the timeout period, it retransmits the data encompassed by the *sliding window*. This process is repeated until the ACK arrives at the sender side.

Multiplexing: Like UDP (see section 2.2.1), TCP supports simultaneous data communication for many processes of a single host. The port concept is used to demultiplex multiple data streams at a receiver. It provides a means to classify data packets and pass them to their corresponding application processes. The tuple of the sender and receiver sockets uniquely identifies a TCP connection between two application processes.

Connections: A TCP connection encompasses the following state information: the end-host sockets, sequence numbers, *sliding window* size. This "connection state" is required to achieve flow control and the other mechanisms described above. The TCP implementation of the processes that wish to communicate must first establish a connection (initialize the status information on each side). TCP supports a *passive* and an *active* mode for connection establishment. A node in passive mode listens on the socket, waiting to accept incoming connection requests rather than attempting to initiate a connection. In order to establish a connection, at least one

of the partners must actively initiate the connection establishment. The active node uniquely specifies the destination host and port of the communication partner. Since connections are usually established between "unreliable" hosts and via "unreliable" networks, such as the Internet, a handshake mechanism with clock-based sequence numbers is used to avoid erroneous initialization of connections. This mechanism is commonly known as *three-way handshake* [DS78].

In Internet communication TCP has been very successfully used for many years. Most applications (for example, WWW and Email) and application support layer protocols (for example, HTTP, FTP, RTSP and SIP) rely on TCP as their transport protocol. However, experience with TCP has shown that early implementations had some drawbacks if used in large-scale environments, such as the Internet. As a result, most modern implementations of TCP contain four intertwined algorithms that improve fault tolerance, resource utilization, efficiency, and scalability [Ste97].

Slow Start Algorithm: According to the TCP version 1 specification [Pos80c], TCP starts communicating by sending multiple packets into the network, up to the window size advertised by the receiver. As a result, intermediate routers must queue packets if there are low-bandwidth links between the sender and the receiver. Since TCP is used within the Internet where thousands of TCP connections are used simultaneously, routers, particularly in the core network, can quickly run out of queue space, and hence, have to drop packets which then have to be retransmitted. This naive flow control approach can reduce the throughput of TCP connections drastically [Pos80b]. The *slow start algorithm* resolves this problem by adapting the rate at which new packets are injected into the network to the rate at which the receiver returns the ACKs.

Congestion Avoidance: The *congestion avoidance algorithm* is the counterpart of the slow start algorithm. The algorithm reduces the packet transmission rate as soon as it detects network congestion. Congestion is assumed if the packet loss rate increases [10]. After the transmission rate is reduce, congestion avoidance then invoke the slow start algorithm to adjust the maximal throughput again. Thus, both algorithms can be deemed to be complementary counterparts.

Fast Retransmission: The idea of the *fast retransmission algorithm* is to retransmit (most likely) missing packets as soon as packet loss is detected without waiting until the retransmission timer expires. Thus, the problem to resolve here is to identify lost packets (or congestion) as soon as possible in order to retransmit these packets very quickly. Congestion or packet loss is indicated by duplicate ACKs being observed at the sender. However, duplicate ACKs might also be caused by packets which

[10] Research on Internet traffic has shown that this implicit assumption of congestion avoidance, namely that packet loss occurs mainly while the network is congested, is reasonable.

arrive out-of-order. The trick is to wait for a small number of duplicate ACKs to be received. If only a re-ordering of the packets occurred, there should only be one or two duplicate ACKs. On the other hand, if three or more duplicate ACKs are received in a row, it is a strong indication that a packet has been lost.

Fast Recovery: After the fast retransmission algorithm has sent the packets that were most likely been discarded by intermediate router, the *fast recovery algorithm* initiates congestion avoidance rather than slow start (which would be normally invoked when congestion occurs). This improvement enables high throughput even under moderate congestion, especially for large windows. The reason for avoiding slow start is that the receipt of duplicate ACKs indicates that data is still flowing between the two end-nodes. Thus, only a moderate or short-term congestion occurred. Preventing TCP from abruptly reducing the flow (which would be the case if slow start is activated) is the trick here.

2.2.3 Real-time Transport Protocol

The *Real-time Transport Protocol (RTP)* provides, according to its specification [S+96], end-to-end delivery services for data with real-time characteristics such as interactive audio and video. RTP is designed to meet the transport requirements of multimedia conferencing applications with two, several, or even a large number of participants.

The real-time transport protocol consists of two closely related parts:

- RTP, the real-time transport protocol, adds flow information of the transmitted real-time media stream to the data packets.
- RTCP, the real-time transport control protocol, provides a feedback channel from the receiver to the sender. It monitors the QoS of the data stream and conveys the QoS feedback along with minimal session control information about the participants in an on-going session.

Even though RTP is supposed to be a transport protocol for real-time streaming applications, it does not provide transport services nor does it guarantees QoS regarding the bandwidth, delay, jitter, or packet loss. It simply adds a protocol header with stream information characterizing the media flow (for example, a sequence number, session id and timestamp) in front of the actual media payload. This information can be used to compute the QoS that a particular data packet experienced on its transmission path. The network QoS estimation can then be fed back to the sender by means of the control protocol RTCP. Sending applications may use the QoS feedback to adapt to the dynamically changing network conditions, for example, by using adaptive encoding, increasing redundancy, and utilizing low-bandwidth encoding formats. The feedback of the momentary transmission quality might also be valuable to diagnose faults and locate whether problems are

local, regional, or global. Moreover, it could also be used by third-party monitors (such as in routers) to monitor performance of the network and diagnose problems. However, since RTCP's feedback mechanism is based on the principles of "closed-loop" feedback, the QoS information will reach the sender after a certain delay. It is therefore not useful for instantaneous adaptation or transmission control.

In more detail reviewed, the basic services provided by RTP are payload type identification, sequence numbering, time stamping and source identification.

The sender timestamps each RTP packet with the relative time (relative to the other samples/packets of the stream) of the first sample of the data stream in the packet. The receiver can use the *timestamps* to reconstruct the original timing before playing the data stream back. They are probably the most important information provided by the RTP header since it is the means to estimate the delay and jitter.

The *sequence numbers* are useful to identify and process packets that arrive out of order at the receiver node. They facilitate also packet loss detection.

The *payload type* is intended to tell the receiving application how to interpret the payload data. Based on the payload type, the receiving application selects the appropriate encoding and compression schemes. So called profiles specify default mappings of payload type codes to payload formats. An initial set of default mappings for audio and video has already been specified in [Sch98]. Besides providing information on how to interpret the RTP payload, the payload type can be used within the network to achieve implicit resource reservations or other QoS guarantees.

The *source identifier* can be used, for example, in audio conferencing applications to indicate the sender (user) currently talking. In multicast applications with several senders, where all sources send their data to the same multicast address, source identification becomes necessary in order to associate incoming packets to the proper data stream.

Figure 2.7 presents the format of the RTP header. Besides the header fields already described, the RTP header includes a version number, padding, extension and marker bits for special or experimental use, the number of contributing sources, and their identifiers. Multiple contribution sources are specified if the payload of the RTP packet contains data from several sources. The synchronization source indicates where the data was aggregated, or determines the source of the data if there is only one.

RTCP, the control protocol of RTP, periodically transmits control packets to all participants in the session. The two main tasks of RTCP can be described as follows:

Delivery Monitoring: RTCP provides QoS feedback by monitoring the receiving data flows and sending those reception statistics or control information back to the senders. Feedback is mainly sent in the form of *Sender Reports (SR)* and *Receiver Reports (RR)*. SRs are issued by receivers that are not only receivers, but also senders. Thus, a SR encompasses all RR information plus additional sender specific transmission statistics, for

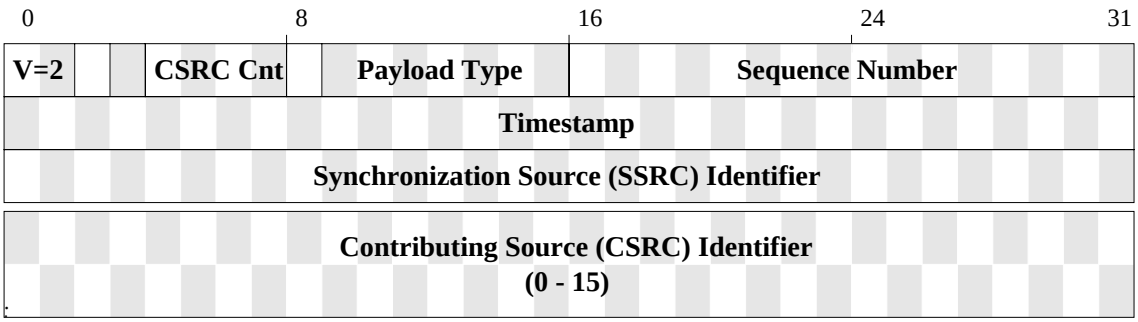


Figure 2.7: The RTP Protocol Header

example, timestamps, the number of RTP data packets, and the number of payload octets transmitted. Both, SR and RR, contain performance statistics giving the number of packets lost, the highest sequence number received, and jitter along with delay measurements to calculate the round-trip delay time.

Identification: RTCP conveys identification information about the participants in an RTP session. This kind of information is transported in a third type of RTCP report called *Source DEScription (SDES)*. The identification information is especially useful for sessions without, or with only "loosely-controlled", session management to distribute minimal session control information to participants that enter and leave the session without being member or negotiating session parameters.

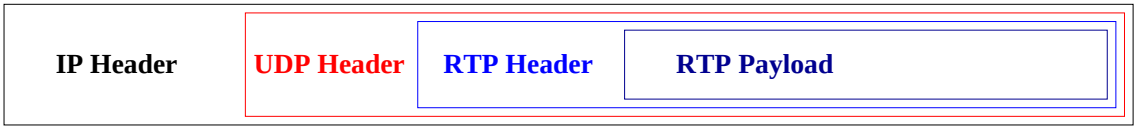


Figure 2.8: IP packet containing real-time data encapsulated in a UDP and RTP packet

Applications typically run RTP and RTCP on top of UDP/IP. RTP, however, is designed such that it can be used with other suitable underlying network or transport protocols. Figure 2.8 illustrates the structure of an IP packet containing real-time data that is delivered via RTP/UDP. Since RTP usually runs on top of UDP/IP, it also supports data transfer to multiple destinations using multicast distribution. RTP and RTCP are usually implemented as part of the application rather than as separate transport level components within the operation system.

As a result, RTP is often criticized to be mistakenly called a transport protocol. And, since RTP provides merely a header containing information about the real-time stream that is attached by the application rather than the transport layer, it is certainly not a transport protocol according to the OSI reference model. Just recently it was proposed

to elevate RTP to the status of protocol, equivalent to TCP or UDP [R+98b]. RTP packets would then be explicitly labeled as such in the IP packet header. Using RTP as a "native" transport protocol has the potential to vastly simplify the problem of classifying real-time streams and header compression. However, since the RTP header carries mainly information relevant to the end-user application (for example, the media payload type or the application timestamp) it would be unwise to declare the application level protocol as transport protocol. A proper transport protocol, for example, does not care for the payload type of a media stream; it might rather be interested in the data rate and peak rates.

2.2.4 Summary

Table 2.2 summarizes the results of this section by comparing the transport services offered by UDP and TCP and the streaming mechanism provided by RTP (RTCP).

Criterion	TCP	UDP	RTP(RTCP)
Reliable transport	+	-	-
– Bit error protection	+	+	+
– Guaranteed packet delivery	+	-	-
– Packet order preservation	+	-	-
Connection-oriented	+	-	-
Packet-oriented	+	+	+
Packet retransmission	+	-	-
Network Rate control	+	-	-
Application rate control	-	+	+
Sequence number	+	-	+
Payload type	-	-	+
Timestamps	-	-	+
Session id	-	-	+
QoS feedback	-	-	+

Table 2.2: Comparison of UDP, TCP and RTP-on-UDP as Transfer Mechanisms

Reviewing the characteristics of UDP and TCP, one can conclude that for normal *(discrete) data traffic* (such as bulk and burst traffic) where end-to-end delay is not critical, TCP is definitely the protocol of choice. The fact that it guarantees reliable transmission makes it superior for non-time-critical data traffic. The successful use of TCP in numerous applications clearly affirms this.

TCP's transport level rate control, namely slow start and congestion avoidance, provide a basis for sharing network bandwidth fairly among network users. By avoiding network congestion, TCP has a significant impact on the utilization of the network in terms of

successful transmission. Although TCP's network rate control is acknowledged as a very valuable protocol addendum, it forecloses application level rate control at the same time. Applications which require short end-to-end delays or need to transmit data with constant bit rates, need the ability to control the transmission rate by themselves. Moreover, time critical or real-time applications perform badly in conjunction with slow-start.

As a result, those applications are better off with the simple datagram protocol, UDP, as a transport protocol. This allows applications to freely control the transmission rate. Furthermore, the lack of reliable transmission might, in the case of time critical applications, be a benefit rather than a disadvantage. Since reliability in the *best-effort* Internet can only be achieved by means of retransmission (if in-band error correction is neglected), which drastically increases the end-to-end delay, retransmitted packets are in most cases worthless, simply due to "too late" arrival. Moreover, applications that require group communications, and hence do directly benefit from the multicast capabilities of the MBone or IPv6, cannot deploy a "connection-oriented" transport protocol.

Even though UDP is currently the better transport protocol for media streaming in the global Internet, it has two serious drawbacks. First, the lack of network-level flow control impedes congestion avoidance. Nothing prevents applications from permanently congesting the network. Second, UDP causes many problems if network resources need to be fairly shared among different protocols and applications. If congestion occurs, for example, TCP backs off whereas UDP keeps sending with whatever rate the application requires. From this follows that TCP traffic is currently suppressed by UDP traffic in the Internet.

Finally, RTP is considered as a streaming mechanism. Since UDP is used on the transport level, RTP-on-UDP has the same transport properties. The additional streaming information, namely the timestamp and session id, can be exploited to compute the "instantaneous" QoS properties of the delivery path. This information is especially valuable if adaptation is deployed within the sender and receiver applications. In order to propagate the QoS feedback to the sender, RTP includes the control protocol RTCP.

Summarizing one can conclude that real-time streaming application which require sender timestamps or can make use of the QoS information clearly benefit from the streaming mechanism RTP-on-UDP.

2.3 Reservation Protocols

Resource reservation protocols generally communicate application QoS requirements to the network elements along the transmission path. If the QoS request is admitted by the network (i.e. bandwidth, processing time, queuing space is at acceptable levels), the resource reservation is established.

A common misunderstanding is that reservation protocols provide better QoS. Those signalling protocols simply establish and control reservations. Enforcement of the reservation

must be provided by another component of the QoS architecture. It is similar to flight reservation systems. The booking system makes sure that a seat is available for a certain passenger by marking the seat as "unavailable" for everybody else. However, if nobody at the airport controls the boarding and checks the flight tickets, the plane might be full of passengers without reservation. Thus, resource reservation protocols (signalling) and QoS control services (controlling) complement each other; but are useless on their own.

This section introduce two known resource reservation protocols that are currently used within the Internet: RSVP and YESSIR.

2.3.1 RSVP

The *Resource ReSerVation Protocol (RSVP)* [Z+93, B+97b] was developed in a joint project at the Information Science Institute of the University of California (ISI) and Xerox Corporation's Palo Alto Research Center (PARC). Today the development of RSVP is carried on in the IETF working groups for RSVP and Integrated Services (see section 3.2.5).

2.3.1.1 Design Goals

RSVP is intended to be a general resource reservation mechanism used within the Internet. It is used to establish reservations for network resources on the path from a data stream source to its destination. The goal of resource reservation is to ensure that the packets are handled within the network such that they meet the QoS demands of the communication applications. According to the specification, RSVP provides "receiver-initiated" setup of resource reservations for unicast and multicast data flows in heterogeneous networks with good scaling and robustness properties.

In the first publication on RSVP [Z+93] the developers listed the main design goals of this new reservation mechanism.

Accommodation of heterogeneous receivers: In a wide are network, such as the Internet, different receivers and the paths to these receivers usually have totally different characteristics. Hence, RSVP shall provide a means for heterogeneous receivers to make reservations specifically tailored to their own needs.

Adaptation to changing multicast groups: The ability to provide communication services to multiple participants at the same time raises another issue; the membership of large multicast groups can be highly dynamic. Thus, RSVP aims to deal gracefully with changes in the multicast group membership.

Exploiting different resource needs for efficient resource utilization: Streaming audio within multicast capable audio conferencing tools generally requires only sufficient network resources for one (or two) audio streams at the same time; it does not

make sense if everybody talks at the same time. Therefore, it is important for RSVP to allow end-users to specify their application needs.

Allowing receivers to switch channels: In a conference with several participants (or speakers), end-users might only be interested in reservations for the media stream of one speaker at a time, but would like the opportunity to switch between various speakers. Therefore, RSVP allows receivers to change "channels" without loosing their reservations. However, this design choice has a significant drawback. If traffic that is not confirm with the reservation is by default sent as simple *best-effort* traffic, nothing prevents this traffic from being sent and wasting a share of the bandwidth.

Adaptation to changes in the underlying network: RSVP relies on the IP routing protocol. In the large internetwork, router errors or overloaded links can force the routing protocol to occasionally re-route data flows. Hence, RSVP should be able to deal gracefully with such changes in routes, by automatically re-establishing resource reservations along the new path if adequate resources are available.

Controlling the protocol overhead: Since scalability is also an important issue in the context of Internet communication, and in particular with respect to multicast communication, RSVP should avoid both, the explosion in protocol overhead when group size gets large, and also incorporate tunable parameters so that the amount of protocol overhead can be adjusted.

Independence of underlying technologies: The last design goal is not specific to the problem of resource reservation. It is a general matter of modular design. RSVP should be designed in such a manner as to be largely independent of its architectural components and the underlying network technology.

2.3.1.2 Operation Overview

Each RSVP capable network node requires several modules; see Figure 2.9 as an illustration.

The inter-operation between modules accomplishes both, reservation setup and enforcement. The RSVP daemon handles all protocol messages required to set up and tear down reservations.

RSVP provides a general mechanism for creating and maintaining distributed reservation state in routers along the transmission path of a flow's data packets. If sufficient network resources are available, its requests will result in resources being reserved in each node along the data path. RSVP only supports reservations for simplex flows, i.e., it requests resources in only one direction. However, nothing prevents an application process from being a sender and a receiver at the same time.

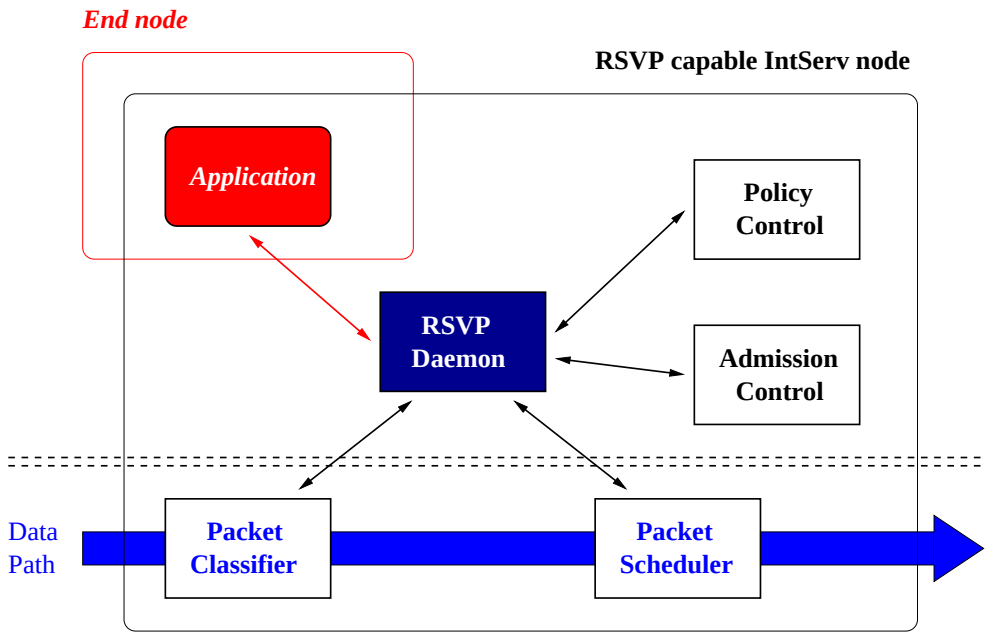


Figure 2.9: Interaction between modules on an RSVP capable node or end host

RSVP is a network level reservation protocol that can be classified as an Internet control protocol similar to the routing protocols. The RSVP process on a network node operates in the background to process the reservation signalling – not in the data forwarding path – as sown in Figure 2.9. RSVP transfers and manipulates QoS and policy control parameters as opaque data, passing them to the appropriate *traffic control* (see section 3.2.5.3) and *policy control* modules for interpretation.

In order to achieve a proper reservation "signalling channel", RSVP deploys "raw" IP datagrams (with protocol number 46) in current implementations. Raw IP datagrams are supposed to receive a better service than standard data traffic due to their network control end. This guarantees that RSVP messages are delivered even under high or overloaded network conditions. Alternatively it is possible to encapsulate RSVP messages in UDP datagrams. This fall-back solution is needed for systems which do not provide support for raw network I/O.

Since reservations in RSVP are receiver-initiated, RSVP must make sure that the reservation messages (RESV) follow exactly the reverse route to the data flow. This reverse path (or tree in the case of multicast) is maintained by periodic path messages (PATH) initiated by the senders. PATH messages are sent "downstream" along the routing path (or tree) provided by the IP routing protocol. Reservation messages (RESV) propagate only as far as to the closest point on the reverse tree where a reservation of equal or greater level for the same flow has already been established. Figure 2.10 illustrates how the PATH and RESV messages travel between the RSVP nodes assuming a simple network topology.

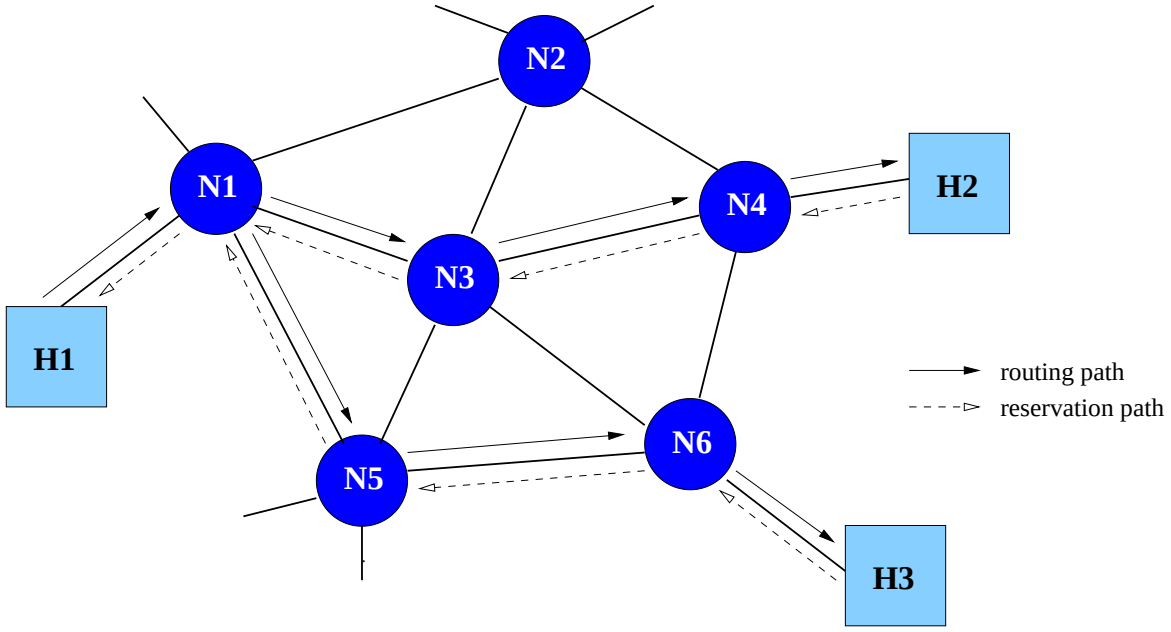


Figure 2.10: A simple network topology with the data path (or tree) from the sender (H1) to the receiver (H2 and H3) and the reverse path (tree) from the receivers to the sender

During the reservation setup phase an arriving QoS request (in an RESV message) must pass two local decision modules, namely the *admission control* that is part of traffic control, and the *policy control*. Admission control checks out whether the node has sufficient resources available to provide the requested QoS. Policy control, on the other hand, determines whether the user has the expected administrative permission to make the reservation. If both subsystems decide to accept the reservation request, the reservation properties are set in the *packet classifier* and in the *packet scheduler* (i.e. in the link layer interface) to obtain the desired QoS. In order to continue the end-to-end reservation establishment along the transmission path, the RESV message is forwarded "upstream" (on the reverse path or tree) towards the sender(s). If the request is rejected, the RSVP daemon returns a reservation error message (RESVERR) to the application which originated the request. When sufficient resources are available, the RESV message will finally arrive at the sender node indicating that the reservation has been successfully established.

Although RSVP is a general mechanism for resource reservation, independent of the QoS traffic control framework, it is so far only used in conjunction with the Integrated Services (IntServ) architecture (see section 3.2.5). The structure and contents of the QoS parameters according to IntServ are specified in detail in [Wro97b]. The parameters for policy control are still under development.

2.3.1.3 Reservation Model and Styles

An elementary RSVP reservation request contains a *flow descriptor*. It basically includes a *FlowSpec* and a *FilterSpec*. The FlowSpec specifies the desired QoS, whereas the FilterSpec, in conjunction with a session specification, defines the "flow" (the set of data) to receive the QoS. The FilterSpec enables QoS guarantees only for an arbitrary subset of the packets in a session. Packets not matched by the FilterSpec are treated simply as *best-effort* traffic.

The FlowSpec includes a service class (currently either controlled load or guaranteed) and two sets of numeric parameters: an *RSpec* which defines the desired QoS, and a *TSpec* which describes the data flow. Both, the format and content of TSpecs and RSpecs are defined as part of the IntServ models [Wro97b] (see also section 3.2.5.2 and 3.2.5.3). The format of the FilterSpec depends upon the underlying network protocol (whether IPv4 or IPv6 is in use). The packet filters operate upon flow information within the packet header, the upper-layer protocol headers or even the payload data. The source and destination IP addresses, the transport ports and the flow label are commonly used to filter a data flow.

Within RSVP three different reservation styles are defined. These are classified in terms of "sender selection" (explicit or wildcard) and "reservations" (distinct or shared) (see Table 2.3).

Sender Selection	**Reservations**	
	Distinct	Shared
Explicit	Fixed-Filter (FF) Style	Shared-Explicit (SE) Style
Wildcard	(None defined)	Wildcard-Filter (WF) Style

Table 2.3: RSVP Reservation Styles

The FF style forces a "distinct" reservation for each individual sender, while SE and WF styles allow the sharing of a single reservation among all packets of the selected senders. The SE style allows the receiver to explicitly specify the set of senders to be included, whereas a WF shares a single reservation with the flows of all upstream senders.

2.3.1.4 Design Principles

In order to meet the design goals presented in section 2.3.1.1, RSVP exploits six basic design principles that are briefly described in this section. These design principles are often used to classify RSVP among other reservation mechanisms and to compare it with other protocols. The principles are described here, and their advantages and disadvantages for resource reservation in the Internet are discussed.

Receiver-Initiated Reservation: In contrast to most other reservation mechanisms, RSVP deploys a receiver-initiated reservation mechanism. Thus, receivers choose the level of resources and decide when to set up and tear down reservations. Since RSVP provides service for multicast applications, senders would have to maintain a reservation for each receiver of the multicast group. This, of course, would not scale for large multicast groups. Thus, RSVP is designed such that the sender does not necessarily need to know the number of receivers and their network characteristics. Furthermore, if network charging is deployed in future networks, the receiver is likely to be the party paying for the requested QoS. Thus, a receiver-initiated reservation mechanism is advantageous.

Receiver-based reservation, however, has two critical aspects. First, in order to allow receivers to reserve resources along the transmission path, which is not known by the receiver due to the lack of routing information, RSVP must deploy the PATH-RESV mechanism described earlier (or some equivalent approach). Second, receivers can mistakenly reserve less bandwidth than the actual data stream requires. The result is that the sender pushes more data into the network than the network can cope with. The consequences are congestion, leading to uncontrolled packet drop, and wasted bandwidth.

Separation of Reservations from Packet Filtering: Unlike most other reservation protocols, RSVP is designed to make a clear distinction between the resource reservation and the filters determining the packets that can use the resources. The packet classifier is responsible for determining the packets that can use the reserved resources. Both, the packet filter parameters and the reservations are set up by means of RSVP.

The main benefit of this separation is that the packet filters can be dynamically changed within a session without changing (and running the risk of loosing) reservations. Thus, the implementation of channel (i.e. sender or speaker) switching becomes simple.

Supporting Different Reservation Styles: Providing reservation styles that support distinct and shared reservations for explicitly selected or all senders of a session offers a flexible reservation architecture. These different reservation styles allow intermediate routers to merge individual reservations for the same session (as in a multicast group).

Maintaining Soft State: RSVP maintains the resource reservations within the individual network elements by keeping them as *soft-state*. Soft-state is simply state information that requires periodical refresh, otherwise it is discarded and its associated resources are freed. Soft-state within RSVP is maintained by periodically sending PATH and RESV messages at configurable refresh intervals. State information is updated by simply sending the new path or reservation states.

The RSVP soft-state mechanism adds both adaptability and robustness. Even if the transmission path changes due to a route change of the network protocol, the soft-state ensures

proper RSVP operation; unused reservations will eventually time out and new reservations will be established along the new path. RSVP also supports explicit tear down messages (TEARDOWN) to avoid holding reservations for several refresh periods longer than required. The soft-state mechanism is also robust against occasional loss of control messages or if hosts lose their entire state due to a crash. The recovery time following such an error depends mainly on the refresh period of the soft-state. The benefits of the soft-state approach, however, do not come for free; The periodic refresh or exchange of messages between RSVP nodes on a *per-session* basis adds significant overhead and does not scale. The periodic messages increase the processing load on core routers and the bandwidth required for signalling increases in proportion to the number of sessions. However, the scalability problem occurs only in the core of the network. If RSVP is used within Intranets or local area networks at the edge of the Internet, scalability is not an issue.

There are several solutions to resolve the scalability problem: first, it is suggested that RSVP is used only at the edge of the network, whereas the DiffServ architecture is deployed in the core (see section 3.2.6); second, research on RSVP has shown that soft-state could easily be replaced by a *hard-state* mechanism with soft-state as fall back solution [MSH98]; and third, ongoing research in the area of aggregation of IntServ state proposes solutions allowing the omission of per-session soft-state in the core network [BV98, GA98].

Protocol Overhead Control: The protocol overhead caused by RSVP is determined by the number of RSVP messages sent, the size of these messages and the refresh frequency. RSVP tries to minimize the protocol overhead by merging path and reservation messages of equal sessions as they traverse the network. Therefore, each link between RSVP nodes carries no more than one PATH message per session in either direction during a path refresh period (and respectively for RESV messages).

Modularity: RSVP mainly interfaces with (1) the end-user application which provides the FlowSpec, (2) the network routing protocol which forwards the PATH messages, and (3) the network admission control process. RSVP is designed to be largely independent of these interfacing components and the FlowSpec format used by the application and the admission control module. The FlowSpec is simply treated as a number of bytes which must be propagated to the admission control modules where it is processed. RSVP is also independent of the underlying routing protocol(s). The only assumption about the routing protocol is that it provides both unicast and multicast routing. RSVP does not assume that the route between a given sender and a receiver is fixed or that it is the same on the reverse path. However, if RSVP obtains notifications of route changes, a fast recovery mechanism, called *local repair* (see [B$^+$97b]), can be deployed to re-establish reservations on the new path. The independence of the route path allows RSVP to be very robust. As long as the network protocol finds a route from the source to destination, RSVP will operate.

In contrast, since changes in the transmission routes are allowed, prevents RSVP from guaranteeing reliable QoS. Route changes due to "new routes becoming available" (rather than "routes being lost") may cause RSVP to lose the reserved resources on the old link if there are not enough resources available on the new path. Since route changes (or oscillations) are not rare in the Internet [11] and RSVP cannot guarantee QoS "promises" when route changes occur, RSVP cannot be classified to be a reliable reservation mechanism.

The routing interface to RSVP [ZK98] is subject to continuous research because more advanced routing facilities are desirable. One research area is "route pinning". Flows should have the option to continue using the "old" path without running the risk of loosing their QoS guarantee if it is still properly functioning. Research in the area of QoS-based routing might also put forth solutions to this problem.

2.3.2 YESSIR

In joint research between IBM's Watson Research Center and Columbia University *YEt another Sender Session Internet Reservations (YESSIR)* mechanism has been developed. YESSIR [PS98] is built on top of RTP, and in particular RTCP (see section 2.2.3). Using RTP to facilitate resource reservation was motivated by the observation that a large proportion of applications demanding QoS guarantees are real-time streaming applications that already use RTP.

Similar to RSVP, as described in 2.3.1, YESSIR uses soft-state to maintain reservation states in the network, supports shared reservations among multiple senders, and is compatible with the Integrated Services architecture (see section 3.2.5).

In contrast to RSVP, however, YESSIR is firstly a sender-initiated and in-band approach to reserve resources in the Internet. Secondly, it aims to be a light-weight reservation mechanism. It requires significantly less code and run-time complexity than RSVP. And thirdly, YESSIR extends classical reservation protocols by allowing operation based on "partial reservations" which are expected to improve over the duration of sessions.

Although both reservation mechanisms can be used as reservation protocol for the Integrated Services architecture, they can operate side-by-side on the same network without interrupting proper operation or affecting each other's reservations.

2.3.2.1 Design Goals

Sender-initiated Reservation: The developers arguments for sender-initiated reservation are contradictory with those of the RSVP designers. First, they do not believe that most applications can make much use of the benefits of receiver-initiated reservations. And second, they believe that sender-based reservation fits better with policy

[11] It has been shown that route changes (or oscillations) occur frequently in the Internet [Pax96a].

and billing, especially since the number of entities making reservations is likely to be much smaller than the number of receivers.

Soft-State: The network nodes maintain YESSIR reservation information as soft-state. Thus, resources are automatically released and state information freed as soon as the periodic refreshes time out. The soft-state makes YESSIR robust against route changes and control message loss. An explicit teardown mechanism is offered to avoid maintaining reservations longer than required.

The fact that route changes cannot be prevented and can only be handled by soft-state also makes YESSIR an unreliable reservation protocol.

Allowing Partial Reservations: Classical reservation mechanisms provide an "all-or-nothing" reservation semantic. Reservations are either granted or denied on a end-to-end basis. YESSIR, in contrast, exploits a reservation model that allows partial reservations where some of the links along the delivery path have resources reserved, whereas others have not. The data flow is simply forwarded as best-effort traffic on those portions where reservation could not be established. Due to the soft-state properties and the periodic reservation messages, links without reservations might acquire a reservation as others tear down. End-users have the choice of allowing operation based on partial reservations or only on end-to-end reservations.

Although partial reservations seems to be a nice addendum, it is still not clear how useful this reservation mode is in practice. Applications use reservation protocols usually to reserve the resources they need for a certain communication session. Therefore, it is arguable whether such applications can be satisfied with partial reservations or if they would prefer not to establish a low-quality communication session.

Providing different Reservation Styles: YESSIR supports also *individual* and *shared* reservations. Individual reservations are established on a per-sender basis while resources for shared reservations are simultaneously used by all senders of an RTP session. Since YESSIR controls (shared) reservations from the sender direction, channel (or speaker) switching is not trivial.

Low Protocol and Processing Overhead: YESSIR is based on in-band signalling where the reservation messages are transported as part of RTCP. Thus, YESSIR does not define another signalling protocol; it merely defines the message objects which are carried in the RTCP sender reports. Reservation establishment requires only one reservation message object.

Since RTCP messages are in small or moderate groups sent by default more frequently than RSVP PATH and RESV messages, the overall protocol overhead may be even worse than in RSVP. Moreover, a proper reservation protocol requires either guaranteed bandwidth for the signalling channel or high priorities for the reservation messages to allow proper reservation control even under high load. In-band signalling built on top of a standard transport level protocol prevents both.

Inter-operability with RTP and IntServ: In order to piggy back YESSIR message objects at the end of the RTCP SR and RR reports, an optional reservation extension for RTCP is defined (see Figure 2.11). Normal RTP operation at the end systems is not affected by this protocol extension. The YESSIR extension includes (a) a generic fragment, defining the desired reservation style, the soft-state refresh interval and whether to make partial reservations; (b) a FlowSpec fragment, determining the QoS control parameters required for admission control and traffic control; (c) an optional networking monitoring fragment, used to store link specific resource information; and (d) an optional reservation error fragment.

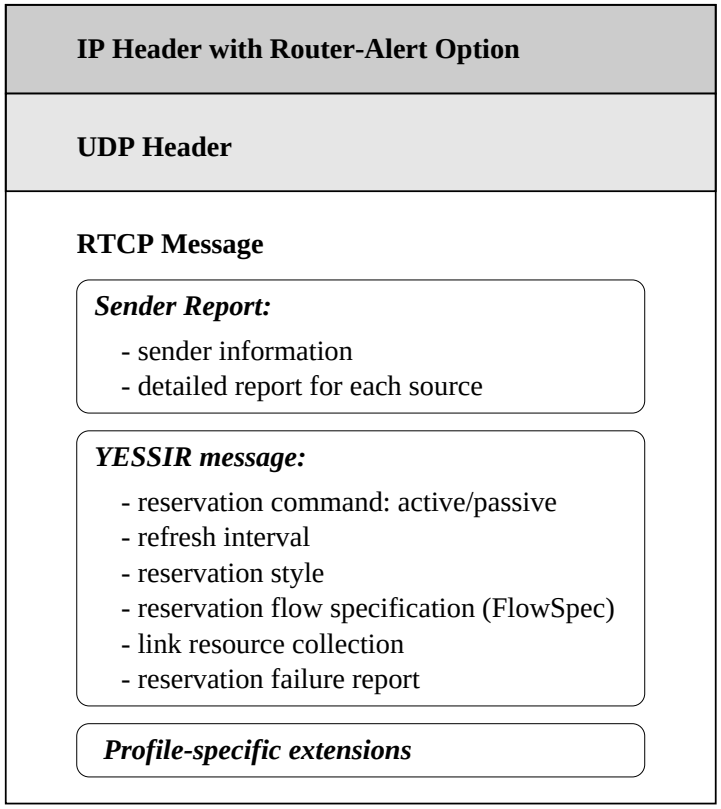


Figure 2.11: The YESSIR message format

YESSIR transfers the QoS control parameters also as opaque data [12] to the network nodes. With this respect, YESSIR is like RSVP independent from the FlowSpec format used by the application and the admission control module. The traffic flow, however, is primarily described in terms of the IntServ models as described in section 3.2.5.3. Besides those FlowSpec formats, two additional formats were considered in the first release: the RTP *payload type* and the IPv4 *type of service*.

[12]The flow specification is treated simply as a number of bytes.

2.3.2.2 Operation Overview

Senders of real-time streaming applications periodically send RTCP sender reports (SR) to the receiver. The SRs include transmission and reception statistics of the data flow transmission. YESSIR inserts the reservation control information into these SRs. In order to notify network nodes that an RTCP message contains important YESSIR messages, the IP router alert option [Kat97, K+98] is used. It indicates network routers to closely examine the content of the IP packet. Intermediate routers that do support the router alert and the YESSIR protocol simply forward the RTCP message unaltered to the next hop. End systems commonly ignore the router alert option.

Deploying the router alert option has the advantage that the reservation protocol can be deployed incrementally. On the other hand, using the IP level alert option for application layer protocol (RTCP) purpose is obviously not a good design decision. Deploying the alert option within YESSIR, for example, forces all routers to closely examine these packets even if a router does not support YESSIR.

If one assumes small size multicast groups, RTCP reports are sent more frequently than RSVP PATH or RESV messages [13]. The refresh period does usually not exceed a couple of seconds. This results in better responsiveness of YESSIR in the case of an error or route change than RSVP. As a result, YESSIR operates well even without a router signal upon a route changes. The better responsiveness of YESSIR, however, comes at the expense of greater protocol overhead.

Finally, since YESSIR is based on RTP, another advantage specific to RTP appears. YESSIR enables reservation establishment only based on the information provided by the RTCP sender report. SRs typically contain a byte count and a timestamp. Based on this information, routers can very easily compute (without having to count all data packets) an estimate of the QoS requirements of the data flow. Thus, YESSIR also provides a reservation service that operates in *measurement mode*.

2.3.3 Summary

The main similarities and differences between the resource reservation protocols RSVP and YESSIR can be summarized as presented in Table 2.4.

From this section one can conclude that YESSIR, although a very simple and light-weight reservation mechanism on top of RTCP that supports partial reservations, has many features in common with RSVP.

It was also shown that YESSIR has a few significant disadvantages. First, building a network resource reservation mechanism on top of an application layer protocol which uses in-band signalling does not allow proper reservation control on highly loaded (or

[13] In RSVP the default refresh period is set to 30 seconds.

Criterion	RSVP	YESSIR
Per-flow reservation	+	+
Soft-state	+	+
Partial reservations	-	+
Channel switching	+	-
Fast error recover	+	-
Reliable	-	-
Initiation	receiver	sender
Signaling channel	out-of-band	in-band
Reservation layer	network	application
Styles	FF, SE, WF	individual, shared
QoS architecture	IntServ	IntServ
Scalable	on edge networks	on edge networks

Table 2.4: RSVP vs. YESSIR: What are the differences?

overloaded) networks. Second, since YESSIR is sender-initiated prevents receivers from choosing the QoS level and determining the set up and tear down of reservations. Third, sender-based reservation also complicates channel switching and limits reservation styles to individual and shared. Forth, even though the refresh period of YESSIR is usually much shorter which has a negative impact on the protocol overhead, the lack of an error recover mechanism delays the reservation recovery in the case of route changes. RSVP which was especially designed for the purpose of resource reservation, resolves these problems in a proper manner.

Although RSVP is currently recognized as the superior resource reservation mechanism, it has two considerable drawbacks in common with YESSIR. First, both mechanisms do not scale in the core of the network due to the periodical refresh on a "per session" basis. Second, they provide only unreliable reservation service since both rely on the underlying Internet protocol and datagram routing.

Current research in the area of resource reservation tries to solve these flaws. A few promising approaches, such as DiffServ and IntServ integration (see chapter 3), session or flow aggregation, and hard-state reservation mechanisms, have already been proposed to reduce the scalability problem. The IPv6 flow label (see section 2.1.2), on the other hand, might have the potential to resolve the "flow routing problem" in future reservation protocols.

2.4 Application Layer Protocols

Among the great number of application layer protocols used in the Internet only the protocols useful for stream setup and control of real-time media streaming applications are

examined in this section.

2.4.1 Hyper Text Transfer Protocol

The *Hyper Text Transfer Protocol (HTTP)* [BL+96] is a generic, stateless, and object-oriented application-level protocol that can be used for a variety of tasks based on the request-response methodology. The light-weight protocol provides service for distributed, collaborative, hypermedia information systems. Since 1990 when the *World-Wide Web (WWW)* initiative decided to use HTTP, it has evolved to one of the most widely deployed application level protocols today.

Although it is currently exclusively used on top of the transport protocol TCP (see section 2.2.2), HTTP is independent of the transport protocol. It merely requires that the underlying transport protocol provides reliable service.

In practice most information systems or multimedia applications require more functionality than simple document retrieval. Therefore, HTTP allows an open-ended set of methods to be used to indicate the purpose of a request.

2.4.1.1 HTTP URLs

HTTP is based on the concept of *Uniform Resource Locators (URL)* [BL+94]. URLs unambiguously identify and locate a resource in the Internet by providing an abstract identification of the resource location. Different URL formats are defined for each major class of Internet service. The first part of the URL specifies the service type, for example, FTP URLs start with `ftp:`, Email URLs begin with `mailto:`, etc.

Although the different URL types have very similar syntax and semantics, from now on the HTTP URL type is emphasized. The HTTP URL syntax is defined as follows:

```
http://<host>[:<port>]/<path>[?<searchpart>]
```

The individual tags are used as follows:

`<host>` specifies the Internet domain name (or IP address) of the server application (if authentification is required, the user name and password can also be encoded)

`<port>` determines the port number of the server process. Well-known protocols have default ports (for example, HTTP uses port 80); hence, it is an optional parameter.

`<path>` determines the location of the resource on the server. It is usually the relative path from the root directory of the HTTP server to the resource. Usually the path is used to specify a single document or script that is invoked by the request. However, the path can also be used to specify other objects (for example, directories, scripts, parameters within the server application) which are invoked, retrieved or changed.

`<searchpart>` provides a mechanism to specify request properties. The `<searchpart>` can be used, for example, to transfer [*variable, value*] pairs to *CGI scripts*[14]. The format is simply: `<varname>=<value>`. The "&" character is used to separate multiple pairs.

An example of a complex HTTP URL is presented here:

```
http://spock.lancs.ac.uk:1090/webaudio?method=PLAY&format=GSM
```

This example shows how URLs can be used to control applications that provide a HTTP interface. The HTTP request is sent to the Internet host `spock.lancs.ac.uk`. The application process listening on port `1090` will receive the request and processes it based on the `<path>` and the `<searchpart>` parameters. In this example *WebAudio*, the real-time audio streaming application developed within this work, is commanded to start streaming the audio source referenced by `webaudio` and to use the `GSM` codec as audio encoder.

2.4.1.2 Overall Operation

HTTP is a classical client-server-based request-response protocol. Clients send the request consisting of the request method, a HTTP URL, and the protocol version to the server. The server then processes the request and responds with a status line including the protocol version and a success or error code possibly followed by the requested resource in the case of a simple retrieval request.

HTTP communication is client-initiated by user agents, such as, Web browsers. A client opens a reliable connection to the server or to some intermediate proxy[15], gateway[16], or tunnel[17] which imitates a HTTP server. The connection provides then a reliable communication channel for the request-response pair. According to HTTP/1.0 each transaction must use a separate connection. Thus, the server closes the connection after having sent the response.

[14] The *Common Gateway Interface (CGI)* defines a general mechanism to invoke scripts or applications as server-side HTTP extensions.

[15] A proxy forwards a request towards the server identified by the URL after applying changes to the request.

[16] A gateway acts like a layer above some other server(s) and translates a received requests to the underlying server's protocol, if necessary.

[17] A tunnel acts as a relay point between two connections without changing the messages; they are used when an intermediary must be passed which cannot understand the messages.

Summarizing, each request/response can be seen as a single transaction between the client and the server. Since the server considers (and indeed sees) successive requests from the same client as being independent from each other, HTTP is said to be stateless.

2.4.1.3 Message Format

This section describes the basic format of the request and response messages according to the HTTP/1.0 specification. In earlier versions of HTTP (i.e. HTTP/0.9), much simpler requests and responses (known as `Simple-Request` and `Simple-Response`) were used. However, version 1.0 demands the more complex `Full-Request` format (see Table 2.5) and `Full-Response` format (see Table 2.6).

A HTTP/1.0 compliant `Full-Request` message sent from a client to a server includes the `Request-Line` and additional option headers fields. The optional `Request-Header` allows the client to pass additional information about the request and the client itself to the server. A detailed format description is presented in [BL+96].

Full-Request	=	Request-Line *(General-Header | Request-Header | Entity-Header) CR[18]LF[19] [Entity-Body]
Request-Line	=	Method SP[20]HTTP-URL SP HTTP-Version CRLF
Method	=	`GET` | `HEAD` | `POST` | extension-method

Table 2.5: Syntax of a full HTTP requests

The `Method` generally used within the WWW is called `GET`. It retrieves the resource identified by the `HTTP-URL` (for example HTML documents, images files, etc.). If the `HTTP-URL` refers to an executable (i.e. a CGI script or application), the output of the application is returned as the response. The `HEAD` method is similar to `GET` except that the server returns only the meta information contained in the HTTP header of the response. The `Entity-Body` is simply ignored. The `POST` method, on the other hand, tells the destination server to accept the `Entity-Body` enclosed in the request as a new subordinate of the resource identified by the Request-URL. This method is especially useful if the `<searchpart>` of the URL becomes too long. In this case the `<searchpart>` should be included in the optional *Entity-Body*, since some systems limit the maximum URL length.

A HTTP/1.0 compliant *Full-Response* message is shown in Table 2.6.

[18]CR = ASCII character decimal 13: *carriage return*
[19]LF = ASCII character decimal 10: *linefeed*
[20]SP = ASCII character decimal 32: *space*

Full-Response	=	Status-Line *(General-Header | Response-Header | Entity-Header) CRLF [Entity-Body]
Status-Line	=	HTTP-Version SP Status-Code SP Reason-Phrase CRLF

Table 2.6: Syntax of a full HTTP response

The `Status-Line` of the HTTP response consists of the protocol version followed by a numeric status code and its associated textual phrase. The `Status-Code` is intended for use by programs whereas the `Reason-Phrase` is intended for the human user. The first digit of the `Status-Code` defines the class of response code (see Table 2.7). The optional `Response-Header` again allows the server to pass additional information about the response and the server itself to the client.

Code	Description
1xx : Informational	Reserved for future use
2xx : Success	Request was successfully received, accepted and processed
3xx : Redirection	Further action must be taken in order to complete the request
4xx : Client Error	Request caused a syntax error or cannot be fulfilled
5xx : Server Error	Server failed to fulfill an apparently valid request

Table 2.7: Categorization of HTTP Status-Codes

The `Entity-Body` is optional in both the `Full-Request` and the `Full-Response`. The `Entity-Header` defines meta information about the `Entity-Body` or, if no body is present, about the resource identified by the request URL. The most important entity header field is the `Content-Type`. It indicates the `media-type` of the data sent within the `Entity-Body`. The `Content-Type` header field is defined as shown in Table 2.8.

Content-Type	=	`Content-Type:` media-type
media-type	=	type `/` subtype *(`;` parameter)
parameter	=	attribute `=` value

Table 2.8: Syntax of the Content-Type header field

Further details about currently specified `media-types` are defined in [RP94]. Two examples are provided to illustrate the use of the `Content-Type` field:

- `Content-Type: text/html`

The data in the `Entity-Body` are textual by nature and in particular HTML encoded. This is the standard content type for HTML documents.

- `Content-Type: application/webaudio`
  The `application` media-type indicates that the data are destined for an external application and in particular WebAudio here. If an application is registered with a certain content-type, the client launches the application upon receipt of the this content-type within the response `Entity-Header`.

2.4.1.4 Shortcomings

An obvious flaw of HTTP/1.0 comes from the requirement that a new TCP connection must be established for every single request-response. This not only makes the performance suffer from the delay introduced by TCP's *three-way-handshake* during connection establishment, but also from the fact that current implementations of TCP deploy the *slow start algorithm* (see section 2.2.2 for more information on TCP). Since slow start is always applied at the beginning of the communication on a new TCP connection, the performance of the request-response delivery suffers greatly. A simple solution to this problem is to support persistent connections which are kept alive for multiple transaction. Moreover, from a network point of view it need to be noted that the shortcoming of establishing a new TCP connection for every HTTP request makes the Web transfer less responsive to network congestion. TCP's congestion control mechanisms are less effective for short-lived connections.

Another problem with respect to the protocol overhead is that HTTP is stateless. The server has no recollection of data types and properties used by the client from one request to another. Therefore, all information must be re-transmitted in every request. This is particularly a problem if HTTP is used for session control within multimedia streaming. Every time a control command is sent, the client must send all information identifying a session to allow the server that maintains the session state can associate the request with the corresponding session. Incorporating a session id as part of the HTTP protocol headers would be very useful (compare the RTSP protocol header section 2.4.2.3).

The performance problem mentioned above was one of the main issues that forwarded the design of the next generation Hyper Text Transfer Protocol (HTTP/1.1) [F+97]. The new protocol makes use of persistent connections between clients and servers. The connection is established prior to the first request and kept alive for the transmission of subsequent requests and responses. The new HTTP protocol also improves its predecessor with respect to the support for hierarchical proxies and caches as well as "virtual hosting" (see also [F+97]).

2.4.2 Real-Time Streaming Protocol

The *Real Time Streaming Protocol (RTSP)* [S+98b] is an extensible framework to control delivery of real-time media data, such as audio and video.

2.4.2.1 Protocol Objectives

RTSP is designed as a signalling protocol for the establishment and control of one or more time-synchronized streams of continuous media. A good comparison of RTSP with a real world device is the VCR remote control. Like a VCR remote control, RTSP can be used to start, stop, and pause selected media clips. This "Internet remote control" supports operation to control both, live data feeds or stored media clips.

RTSP is often misunderstood to be a transport protocol. However, it is not involved in the delivery process of the continuous streams itself. It provides a means to negotiate the transport mechanism that should be for the media delivery. RTSP itself is independent of any particular transport mechanism. All current Internet transport mechanisms, namely UDP, TCP, and RTP-on-UDP are supported. The signalling channel of RTSP is also independent of the transport protocol; both UDP and TCP are supported.

The basic protocol design is very similar in syntax and operation to HTTP/1.1 (see section 2.4.1). The benefits of this design decision are: first, many mistakes made in HTTP could be avoided, and second, approved features of the HTTP implementation and the extension mechanisms could be re-used.

In addition to the design properties described so far, RTSP can be characterized under the following design features:

Multi-server capable: Streams of different media servers can be controlled simultaneously. Synchronization is performed at the application level, for example, by means of RTP.

Proxy and firewall friendly: Since RTSP has inherited the protocol format of HTTP, only a few simple modifications to proxy and firewall systems enable the proper handling of RTSP signalling within these systems. In addition, by parsing the SETUP method, firewalls could easily find out the transport ports used by the media streams and open "gates" for the respective media traffic.

Supports load balancing: RTSP can redirect requests to achieve load balancing on the media servers.

Capability negotiation: A priori negotiation of the supported capabilities can be deployed.

Secure: RTSP can make use of the Web security mechanisms (for example, HTTP authentication).

Extendable: New methods and parameters can be easily added to the protocol.

Easy parsable: Only few changes are required to make standard HTTP parsers RTSP compatible. This is also due to the similarity of the protocol formats.

Presentation description neutral: The protocol is not fixed to a particular presentation description format.

High time accuracy: RTSP is suitable for professional applications (for example, remote digital editing) due to support of time stamps with frame-level accuracy.

The main functionality offered by RTSP for media session control can be summarized as follows:

Retrieval of media data from media server:

During start up clients may request a presentation description via HTTP or some other mechanism to get information about the available media streams (if not already known). The presentation description file a specification of the individual media streams (identified by RTSP URLs) including their encodings, language, and other parameters that enable the client to choose the most appropriate combination of media streams. In the case of multicast media streams the presentation description includes the multicast addresses of the presentation. Otherwise, if the media presentation is sent via unicast, the client provides the destination (for security reasons). Based on the presentation description, clients can initialize and control the presentation using the adequate RTSP requests.

Invitation of a media server to a conference:

A media server can be "invited" to join an existing conference in order to play back additional media or to record the media streams of a ongoing presentation.

Addition of media to an existing presentation:

RTSP servers are capable of notifying clients if new media streams are becoming available. This is especially useful for live presentations.

2.4.2.2 Methods and States

Sophisticated stream control requires operations to setup, play, record, pause and stop media streams. Since some control operations, such as play or record are not momentary by nature, but require continuous processing, the RTSP server must maintain session state. Server-side session state offers also a means to check the validity of control requests. A pause control request, for example, is only sensible if the stream is either played or recorded.

The RTSP server generates a `session id` which is assigned to a new session during the initial setup. The `session id` serves as a unique identifier for the session and is used to reference the newly allocated session state within the server. All subsequent RTSP requests and responses include the `session id` as an session identifier in their protocol header. Upon an RTSP request or response, the server or client can easily identify the session to which the control request or response is associated.

RTSP methods which have an impact on the server-side state of a stream described here:

`SETUP` causes the server to allocate resources for the session state. If successful, RTSP changes to *Ready* state; otherwise, it keeps the initial *Init* state.

`PLAY and RECORD` starts data transmission or reception of a stream already initialized via SETUP. If successful, the RTSP state switches to either *Play* or *Record* state.

`PAUSE` temporarily pauses a stream without freeing the stream's state resources. It causes the state to change back to *Ready* again.

`TEARDOWN` finishes an RTSP session and frees resources associated with it. Afterwards RTSP remains in *Init* state.

Table 2.9 gives a brief overview of all methods specified within RTSP.

2.4.2.3 Message Formats

RTSP requests contain mainly the method which should be applied to the resource, the identifier of the resource (`RTSP-URL`), and the protocol version. See Table 2.10 for the complete message format:

RTSP URLs

RTSP uses so called RTSP URLs to uniquely identify presentations descriptions or media stream within the global Internet. The format is similar to the format of HTTP URLs (see section 2.4.1.1).

```
(rtsp|rtspu)://<host>[:<port>]/<path>
```

According to the RTSP URL format, RTSP supports two basic URL types. The first type, namely `rtsp`, requires that RTSP signalling (not the data stream) is issued via a reliable transport protocol (currently only TCP in the Internet) while the second type, namely `rtspu`, specifies an unreliable protocol (currently UDP). If no port is specified, the default RTSP port 554 is assumed.

An example RTSP URL might look like:

Method	Description	Direction	Req.
OPTIONS	Requests a list of the supported of the RTSP service	C ↔ S	required
SETUP	Initializes a new RTSP session (e.g. defines transport mechanism); also used to change transport parameters	C → S	required
PLAY	Starts the playback of a media stream	C → S	required
TEARDOWN	Stops the stream delivery and frees the resources of the stream state	C → S	required
PAUSE	Halts the delivery of the media stream, but keeps the session state	C → S	recomm.
DESCRIBE	Retrieves the presentation description identified by the request URL	C → S	recomm.
RECORD	Starts recording a media stream	C → S	optional
ANNOUNCE	Posts the description of a presentation to a server or updates a session description at the client	C ↔ S	optional
REDIRECT	Redirects a client to a different server	S → S	optional
GET_PARAM	Retrieves the value of a presentation or stream parameter	C ↔ S	optional
SET_PARAM	Set the value of a presentation or stream parameter	C↔ S	optional

Table 2.9: Overview of RTSP methods, their direction and requirement

Request	=	Request-Line *(General-Header | Request-Header | Entity-Header) CRLF [Message-Body]
Request-Line	=	Method SP RTSP-URL SP RTSP-Version CRLF
Method	=	`GET` | `HEAD` | `POST` | extension-method

Table 2.10: Syntax of an RTSP request

```
rtsp://audioserver.comp.lancs.ac.uk:1554/liveaudio
```

Like the RTSP request, the RTSP response (see Table 2.11) is also very similar to the HTTP response format. Besides the `RTSP-Version` some `Status-Codes` were changed to express the failure of RTSP processing (see [S+98b] for details). The categorization of the status codes is identical to HTTP as shown in Table 2.7.

Response	=	Status-Line *(General-Header | Response-Header | Entity-Header) CRLF [Message-Body]
Status-Line	=	RTSP-Version SP Status-Code SP Reason-Phrase CRLF

Table 2.11: Syntax of an RTSP response

An example of a SETUP request and its response in the case of a successful operation is shown here:

Request (C → S):

```
SETUP rtsp://audioserver.lancs.ac.uk/liveaudio RTSP/1.0
CSeq:  302
Transport:  RTP, unicast, client_port=4588-4589
```

Response (S → C):

```
RTSP/1.0 200 OK
CSeq:  302
Transport:  RTP, unicast, server_port=6256-6257
Session:  234303923
```

2.4.2.4 Relations to Other Protocols

This sections discusses the relationship between RTSP and several other protocols that interact with RTSP.

RTSP clearly distinguishes between stream control and conference initiation. Although it supports invitations of servers to play (or record) a stream into (of) a presentation, RTSP is no means intended to be a conference initiation protocol. In order to invite an RTSP server to a conference, session protocols like the *Session Initiation Protocol (SIP)* [H+98] or *H.323* [H.396] can be used. Both are control protocols to setup, maintain or terminate multimedia conferences with one or more participants.

As mentioned earlier RTSP has much in common with HTTP and especially with respect to the protocol format and basic operation. It may also interact with HTTP such that the stream presentation description is retrieved by means of HTTP rather than using the DESCRIPTION method recommended by RTSP. The close relationship to HTTP benefits RTSP requests when they pass proxies, firewalls, tunnels and caches. They can be processed similarly to HTTP requests as described in [F+97].

On the other hand, RTSP differs in several important aspects from HTTP:

- RTSP servers maintain session state information unlike HTTP which is stateless by nature.
- RTSP is a symmetric protocol where both, clients and servers can issue requests, whereas HTTP is asymmetric (only clients issue requests).
- In RTSP data streams are carried out-of-band by a different protocol (for example, UDP, TCP, or RTP/UDP). HTTP, in contrast, carries the payload in-band.
- RTSP provides a different and a more comprehensive set of methods.

Furthermore, RTSP is not tied to any specific transport protocol. Although studies have shown that most real-time media streaming applications use RTP-on-UDP as a transport protocol, UDP or TCP can also be deployed as data delivery mechanism.

Finally, RTSP requires a presentation description format which can express both static and temporal properties of a presentation containing several media streams. The description format of choice here is the *Session Description Protocol (SDP)* [Han98]. It is especially designed to describe multimedia sessions for the purposes of session initiation and control.

2.4.3 Summary

This section compares the application level protocols HTTP and RTSP and discusses their usability as stream control protocols. Table 2.12 summarizes the main similarities and difference of these protocols.

Criterion	HTTP	RTSP
Extensible protocol	+	+
Transaction-oriented	+	+
Flexible request format	+	+
Extension mechanisms	+	+
Symmetric	-	+
Maintains state	-	+
Redirect requests	-	+
Transport protocol	TCP	TCP, UDP
Client available	+	-

Table 2.12: HTTP or RTSP: How useful are they for session control?

Although HTTP is mainly used within the WWW for the purpose of information retrieval, it can also be used for stream control ends. The fact that HTTP requests are freely extendable by means of the `<searchpart>` mechanism (see section 2.4.1.1), stream control operations that are not really supported by HTTP, such as play, pause, stop, etc., can be

emulated. The following example shows how simple RTSP requests can be emulated by HTTP.

RTSP request:

```
PLAY rtsp://server/audio RTSP/1.0
Range:  npt=0-
...
```

HTTP request:

```
GET http://server/audio?METHOD=PLAY&RANGE=npt=0- HTTP/1.0
...
```

The lack of the notion of a session within standard HTTP makes the mapping between single HTTP requests and the session to which the control request should be applied more tricky. One simple mechanism is to use the source IP address to identify the session. This, however, limits the stream control to a single session per IP address. The transport port of the connection cannot be used since usually multiple connections are establish during the lifetime of a session. Alternatively the client could simply add the transport port to every RTSP request. RTSP, on the other hand, uses a `session id` field in the protocol header which uniquely identifies the session.

Another difference between HTTP and RTSP is that HTTP is limited to use the reliable transport protocol TCP within the Internet whereas RTSP is independent of the transport protocol. It can deploy UDP for its request-response messages instead. Correlation ambiguities among request-response pairs are resolved by means of the message sequence counter called `CSeq`.

The main advantage of RTSP over HTTP is that RTSP is a symmetrical protocol. Thus, the server can also initiate communication. This enables RTSP to invite media servers and to add media streams to active presentations. Both operations require the server to actively send a request and thus cannot be done with HTTP. Server-initiated communication also enables server redirection. Servers are capable of forwarding a client request transparently to another server. This capability facilitates server-based load balancing.

Even though this comparison shows that RTSP is clearly the more sophisticated stream control protocol, and hence, preferable in cases where the whole range of control functionally is required, simple HTTP-based stream control has the advantage that existing HTTP user agents (such as standard Web browsers) can simply be used as clients. Basic control functionality can very easily be encoded in HTTP URLs.

2.5 Summary

This section concludes the study on Internet multimedia protocols presented throughout this chapter. The study examines the most important protocols currently used within Internet multimedia applications and explores their usability within interactive real-time media streaming in the Internet.

The network layer protocol IPv6 is compared with its predecessor IPv4. Besides resolving the address problem, the benefits of the new Internet protocol regarding interactive real-time streaming can be summarized as follows: (1) IPv6 improves the packet processing in every intermediate router due to simplification of the IP header. This has the potential to decrease the overall network load and to reduce the end-to-end delay of real-time streaming simply due to faster packet processing in each intermediate router. (2) IPv6 provides native multicast and security support which facilitates media broadcast and secure data transmission within streaming applications. (3) The IPv6 flow label, which introduces the concept of a flow, resolves the implicit layer violation problem of RSVP and has a great impact on the performance of packet classification (see section 5.3). Although IPv6 has the potential clearly to improve real-time streaming applications, it does not magically resolve the QoS problems of Internet communication.

The transport protocols and streaming mechanisms discussed are: UDP, TCP and RTP-on-UDP. Whereas TCP is not suitable for interactive real-time streaming applications because of the interference of its congestion control and reliability mechanisms with the requirements of time-critical applications, UDP provides a simple but sufficient service. RTP-on-UDP is recommended as the streaming mechanism for real-time applications because it adds valuable stream information to the media packets. RTP's control protocol RTCP is especially useful in conjunction with server-site adaptive mechanisms due to the QoS feedback channel. Even though RTP provides only a protocol header for application level streaming information – therefore misleadingly called a transport protocol – it is still useful since most real-time streaming applications require this information within each packet. Thus, the application level protocol assists application developers in designing media streaming applications, and more important, it provides a generic or standardized streaming mechanism which encourages inter-operation of different applications.

The resource reservation protocols RSVP and YESSIR are discussed. Even though RSVP is currently the resource reservation protocol of choice within the IETF, it has several significant drawbacks: (1) The periodic PATH and RESV messages and the per flow PATH and RESV state within network routers do not scale successfully in the core of the network. (2) Due to the lack of acknowledge messages, RSVP has a very slow establishment time if initial PATH or RESV messages are lost. (3) The protocol overhead due to the periodic messages which are required to maintain the soft-state is significant. (4) RSVP is not a reliable reservation protocol, since it is dependent on the underlying routing protocol. An "unnecessary" route change, for example, might cause an application to lose its reservation. The simple and light-weight reservation protocol YESSIR, in contrast, is not a network level

reservation protocol. The signalling is simply based on top of the application level protocol RTP/RTCP and hence can only account for very limited reservation services. The lack of a routing interface prevents YESSIR from quickly establishing new reservations when route changes occur. A reservation along the new path is set up based only on the periodic refresh messages. Since the feedback period of RTCP for small groups is very short, the protocol overhead is significantly higher than in the case of RSVP. As a result, RSVP is recommended as the resource reservation protocol within real-time streaming applications.

The application level protocols HTTP and RTSP are discussed and evaluated for use in stream control. RTSP is a sophisticated stream control protocol with similar functionality to a "VCR remote control". It provides sufficient control functionality for stream control and is therefore recommended for use within media streaming applications. The stream control functionality of RTSP is clearly separated from the session description, as provided by SDP, and the session initiation, as supported by SIP or H.323. It is, however, intended to be used in conjunction with these protocols. Whereas RTSP based stream control requires special RTSP capable clients, HTTP based stream control can simply be accomplished by means of standard Web browsers. The standard Web protocol HTTP, however, can only account for simple stream control. The lack of session or stream semantics limits its usability. Although HTTP could be easily extended in order to provide equivalent functionality to RTSP, it would then lose its main advantage, namely that stream control can be achieved simply by means of a standard Web browser. Since RTSP is designed similarly to HTTP, an RTSP control interface can be easily extended to provide a service also for the simple control based on HTTP.

One can conclude that current real-time audio streaming applications should be developed considering the following design recommendations:

- IPv6 has several benefits compared to IPv4 with respect to network QoS and group communication.
- UDP should be used as transport protocol.
- RTP is a useful application level protocol which facilitates media streaming. In conjunction with its feedback mechanism RTCP, RTP encourages adaptive mechanisms based on QoS feedback.
- RSVP, the network level resource reservation protocol, is preferred over the lightweight application level reservation mechanism YESSIR for resource reservation.
- RTSP is a comprehensive stream control protocol. HTTP is sufficient for simple stream control. Both protocols can easily co-operate within a multi-protocol interface.

Chapter 3

Real-Time Streaming in the Internet

This chapter discusses various application level techniques that are used to compensate for the lack of network QoS and explores current network level QoS models which aim at improving the usability of real-time media streaming applications within the Internet.

Current Internet researchers have not yet come to an agreement on how to achieve QoS for these applications in the network. Some believe that the QoS in the network depends only on the amount of available bandwidth. Hence, they propose simply to increase the amount of bandwidth to resolve the QoS problems. Temporary bottlenecks or momentary service degradations could be overcome by means of *adaptation* (see 3.1.3). On the other hand, others think that the network should rely on resource management mechanisms in order to guarantee QoS to the user. Resource reservation is one mechanism to achieve guaranteed service for real-time media streaming applications.

Section 3.1 describes several application level techniques to improve the quality of simple *best-effort* based network communication. Section 3.2 network level QoS mechanisms that are being currently discussed.

3.1 Application Layer QoS

This section describes several application level mechanisms that are worth considering when developing real-time streaming applications for the Internet. These techniques are designed to achieve higher quality application services to increase end-user satisfaction.

3.1.1 Packet Transfer

With respect to packet transfer, interactive real-time streaming applications have a choice of transport mechanisms, packet sizes and packet transmission techniques that are considered here.

Choice of Transport Protocol

Among current transport protocols in the Internet, namely UDP (see section 2.2.1) and TCP (see section 2.2.2), the former should be used when real-time behavior is required. According to the discussion in section 2.2.4 TCP has several drawbacks for use with time-critical traffic. First, to achieve the *reliability* provided by TCP, retransmission and buffering are unavoidable. Both techniques, however, introduce additional and intolerable delay. Second, real-time media streaming applications need to control their transmission rate rather than leaving this responsibility to a transport protocol that is not aware of QoS. The *slow start* algorithm, for example, prevents applications that use TCP from transmitting with the data rates dictated by the live media sources.

As a result current real-time streaming applications use primarily the simple connectionless protocol UDP as their transport protocol. It allows application to freely control their transmission rate. Moreover, if real-time streaming applications want to benefit from the multicast capabilities of the MBone or IPv6 for group communication, they are currently limited to the use of UDP as their transport protocol.

In the case of real-time streaming the streaming mechanism of choice is currently RTP-on-UDP (see section 2.2.3). RTP on top of UDP adds useful streaming information, such as timestamps, session id, sequence numbers, etc., to the data packets. RTP's feedback channel, namely RTCP, can be used to send QoS information back to the sender. The QoS feedback is especially valuable if adaptation (see also section 3.1.3) is deployed within the sender applications.

Packet Size

Besides the appropriate choice of transport protocol the packet size used to transmit media streams plays an important role with respect to the end-to-end transmission delay.

In the case of audio or video streaming the payload size is usually a multiple of the media frame size. Audio, for example, is read from the audio device in chunks called audio frames. The number of audio samples contained within an audio frames depends on the device configuration. If the end-to-end delay must be minimal, as in the case of interactive real-time applications, the frame size should be small and the payload should encompass as few frames as possible in order to minimize the packetization delay. Typical live audio applications use frames that encompass 20 or 40 ms of audio data.

The $FrameSize$ is determined by the audio format captured from the sound device. Equation 3.1 shows how to calculate the $FrameSize$ in general. For example, the encoding of 40 ms of 4 kHz monophonic audio[1] in 16 bit samples results in a $FrameSize$ of 640 bytes.

[1]Sampling of 4 kHz audio demands a sampling rate of 8 kHz.

$$FrameSize = Channels \times SampleSize \times SampleRate \times Time \quad (3.1)$$

The packet payload size is determined by the number of frames sent per packet and the compression or encoding scheme (see equation 3.2). For example, if we consider packets carrying GSM encoded audio data of one mono, 16 bit PCM frame, the payload size results into 65 bytes.

$$PayloadSize_{GSM} = Frames \times (CompressionRate \times FrameSize) \quad (3.2)$$

In general, if sophisticated compression algorithms such as GSM, CELP, MPEG, etc., are used to encode the media data, a payload size of only a few media frames is very small. Transferring data packets with a payload size of size less than 100 bytes seems to be inefficient considering that the packet headers in the case of UDP/IP or RTP/UDP/IP transport is already of great size. Equation 3.3 presents the formular to calculate the packet overhead.

$$Overhead = \frac{HeaderSize}{PacketSize} = \frac{\sum_{Protocols} Size(Header)}{\sum_{Protocols} Size(Header) + Size(Payload)} \quad (3.3)$$

Table 3.1 shows the packet overheads above the link layer of one mono 16 bit PCM frame using uncompressed and GSM encoded audio. This illustrates that the overhead is highly dependent on the transport and application protocols used. Although the packet overhead for small audio packets is very high, the price must be paid if the end-to-end delay must be minimal.

Transport Mechanism	Total	PCM		GSM	
	HeaderSize	*PayloadSize*	*Overhead*	*PayloadSize*	*Overhead*
UDP/IPv4	28 bytes	640 bytes	4%	65 bytes	30%
RTP/UDP/IPv4	44 bytes	640 bytes	6%	65 bytes	40%
UDP/IPv6	48 bytes	640 bytes	7%	65 bytes	42%
RTP/UDP/IPv6	64 bytes	640 bytes	9%	65 bytes	50%

Table 3.1: Packet Overheads of different Audio Encodings and Transfer Mechanisms

To minimize *Overhead*, or in other words to maximize the network utilization, research in the area of header compression [D+98b] suggests to transmit only the header information subject to changes (for example, sequence numbers, timestamp, checksum, etc.). Thus, the packet *Overhead* can be highly reduced.

Traffic Shaping

Another technique used to improve transmission quality with respect to packet transport is called traffic shaping. The general principle of traffic shaping is to alter the traffic characteristics of the outgoing stream of packets (or cells) on the virtual connection in order to optimize the use of network resources.

In the past traffic shaping played an important role in the research of ATM cell switching. The main schemes proposed by the ATM Forum and ITU are based on the Generic Cell Rate Algorithm and are associated with the Constant Bit Rate (CBR) and Variable Bit Rate (VBR) services.

In the Internet, as it is perceived today, packet losses are of bursty nature [B+97a] due to queue overflows in end systems or intermediate routers. Therefore, applications for live media streaming should shape the outgoing stream of packets such that the data packets are transmitted isochronously over time rather than in bursts. Although fewer packet losses are expected when applications apply traffic shaping to their data traffic, traffic shaping within network nodes is much more effective.

3.1.2 Forward Error Correction

Forward Error Correction (FEC) developed for packet media streaming is a different approach to address the problem of packet loss within Internet communication.

While *closed loop* mechanisms, such as *Automatic Repeat Request (ARQ)* mechanisms, are used to recover lost packets by means of retransmission, *open loop* mechanisms, such as FEC mechanisms, transmit redundant information with the original media data so that lost packets can be restored from the redundancy. ARQ mechanisms, however, are not suitable to recover from packet loss if real-time data are streamed in wide-area networks, such as the Internet. Retransmission introduces too much latency. In addition ARQ mechanisms do not scale well when multicast is used.

Research at INRIA [BC95, BVG97] found that open loop error control mechanisms based on FEC are adequate to reconstruct most lost packets of a media stream if packet losses are isolated to some extent.

Most FEC mechanisms proposed in literature suggest sending "redundant" packets every nth packet which is obtained by *exclusive-or*ing the other n packets [SM90]. This mechanism allows to recover a single loss in an n packet message. It increases the data rate r only by $\frac{r}{n}$. But if a lost packet needs to be recovered, the receiver must wait for the next "redundant" packet. Thus, the mechanism significantly increases the average end-to-end delay when losses are frequent.

The FEC mechanism developed at INRIA especially addresses the problem of consecutive packet losses. The idea is to add highly compressed copies of the previous k frames to the

current media stream packet. If packet n (where n is the packet's sequence number) is lost, any of the following packets $n+1$, $n+2$, ... or $n+k$ must be successfully received. Thus, up to k successive packet losses can be tolerated since low quality substitutes are available. The optimal choice of k is a trade off between the momentary probability of consecutive packet losses in the network and the available bandwidth. The increase of the data rate caused by the redundant information depends on the compression schemes. If low-bandwidth encoders are used for the "redundant" frames, the data rate increases only slightly even if k is set to several frames. However, introducing too much redundancy increases also the data rate which in turn might have a negative impact on network congestion and thus reliability.

In the case of real-time streaming the maximum value for k is mainly limited by the maximum end-to-end delay boundary. In addition to the processing delay of the FEC encoding and decoding, this mechanism adds at least k frame delays to the end-to-end delay. The value for k must be chosen carefully in order to prevent the additional delays from causing the recovered packet to arrive too late and thus being "lost" anyway. This approach has the advantage that the additional latency at the receiver depends entirely on the amount of consecutive packet loss. If no packet loss occurs, no additional delay is introduced. If only a single packet is lost, the recovery can be achieved as soon as the next packet arrives. Thus, receiver latency increases with the number of successive losses.

Summarizing, one can conclude that FEC is an effective alternative to ARQ for providing reliability with only a small increase of end-to-end delays when lost packets are isolated. The effectiveness depends on the characteristics of the packet loss process of the network.

An analysis of packet losses in the Internet [BC95] shows that the probability of consecutive packet losses decreases with the number of lost packets in a sequence. Experiments have shown, for example, that about 90% of the losses are in the range of 1 to 2 consecutive lost packets, whereas about 99% have less or equal than 3 consecutive lost packets. Furthermore, the analysis shows that the probability for consecutive packet loss is highly correlated with the current network load. For example, the average loss gap measured on a 65% loaded network was about 1.2 packets, whereas on a network with 90% load an average loss gap of about 2.8 packets was measured [BC95].

As a result one can conclude that under low or moderate network load, FEC operates well since packet loss occurs infrequently. Even under heavy network load, when consecutive packet losses appear, FEC is capable of recovering most lost packets when multiple frame redundancy is used. Since the likelihood of consecutive packet losses decreases with the number of lost packets, FEC results in good performance. Furthermore, the effectiveness of FEC can be highly improved if used in conjunction with traffic shaping (see section 3.1.1). Both mechanisms are considered complementary since traffic shaping has the potential to "isolate" packet losses whereas FEC resolves the problem of "isolated" packet loss.

3.1.3 Adaptation

Although recent developments within the IETF focus on QoS issues and discuss how to provide QoS for packet-switched networks, such as the Internet, none of the QoS mechanisms are widely deployed. Thus, today's media streaming applications must still tolerate variations in QoS (i.e. dynamic changes of delay variations, preserved throughput and packet loss) delivered by the network.

Mechanisms to provide continuous service even when external conditions change (i.e. network congestion, router queue overflows and processing overload) are commonly known as QoS adaptation mechanisms. Adaptive applications are able to gracefully adapt their service quality depending on the QoS received from lower-level services. Even severe service fluctuations can be accommodated by means of QoS adaptation mechanisms. In such cases, however, it is often appropriate to inform the application of the service degradation so that it can adjust to the new QoS level [CCH92]. If the delivered performance violates the negotiated QoS (for example, QoS reserved by means of RSVP), the user may choose to take some remedial action (i.e. adjust application state to accommodate the current load conditions, re-negotiate the flow's QoS, disconnect from the service).

Application level QoS adaptation is mainly increasing or reducing the QoS properties of the application depending on variations in the network QoS characteristics. Adaptation, for example, changes the media stream (i.e. audio quality, encoding format), adds redundancy to the stream, or adjusts the receiver buffer size (i.e. playout point estimation) to make users think that their application have constant network service qualities. This trick, however, works only as long as the "real" QoS provided by the underlying network is within a certain range. If the "real" QoS degrades below the "adaptation limits", adaptation cannot operate properly anymore and the quality remains poor. An example of QoS control for adaptive distributed multimedia applications is given in [GS95].

If the network supports QoS by means of resource reservation, applications have "guaranteed" resources for their media stream, and hence, need not adapt to changing network QoS characteristics. Thus, in an environment where resource reservation is available, adaptation is only of user-initiated nature.

In a recent comparison of *best-effort* versus resource reservation [BS98], it is pointed out that adaptation mechanisms enable multimedia applications over simple *best-effort* networks to perform – from a user's point of view – very similarly to applications that make use of resource reservation. The *utility* or value that users derive from adaptive application under moderate QoS conditions is almost equal to rigid applications with resource reservation support. Since adaption accounts only for limited service fluctuations, the remaining question is to what extent applications can adapt; the comparison did not find an answer.

In summary we can conclude that *adaptive applications* significantly improve performance under moderate to high network load. However, they can only account service degradation to a certain level.

3.1.4 Receiver Buffering

Since the quality of real-time media depends mainly on timely delivery and play out of the stream data, protocols and mechanisms must address the control of delay, jitter and reliability in an integrated fashion.

Receiver buffering is required to compensate for delay variations, also called jitter, introduced by the network and the processing in the end systems (see section 1.2.6). It is also important to resolve the problem of erroneous data transmission, such as packet reordering. The tasks of receiver buffering are to estimate the optimal *playout delay*, which is required to compute the buffering time, and to manage the queuing of the received packets until their playback point exceeds. The playback time for each packet is usually determined by the timestamp assigned by the sender and an estimate of the network and processing delay.

$$T_{Playback} = T_{Recording} + D_{Network} + D_{Processing} \tag{3.4}$$

The processing delay estimate $D_{Processing}$ accounts only for the processing delay (i.e. decoding, decompression, scheduling) at the receiver. Since receivers cannot differentiate the delay (or jitter) caused by the sender processing and the network, the network delay estimate $D_{Network}$ determines the packet delay up to the receiver.

The playout delay can be constant throughout the entire session or can be adjusted adaptively during the session. Since end-to-end delays in the Internet vary significantly over time [Bol93, S+97], a constant, and non-adaptive playout delay estimation performs badly. Adaptive playout adjustment can be accomplished on a "per-talkspurt" or a "per-packet" basis. In the context of packet audio the playout delay estimate for the first packet of a talkspurt[2] is crucial since it determines the playback time for all subsequent packets of the talkspurt. It should be noted that playback gaps in the audio signal are immediately recognized and perceived as disturbing crackles (compare section 1.3.1.3). In the case of video streaming playout delay adjustment can be done on a "per-packet" basis if the video and the audio are coupled only loosely, or if the video is played on its own. Human image recognition does not notice small variations in the display time of individual frames.

As an example, the problem of playout estimation in the case of packet audio streaming is discussed here. Figure 3.1 illustrates the problem of delay variations caused by the network. The playout delay of the i-th packet dp_i is the sum of the network delay d_i and the buffering time b_i.

$$d_i \;=\; a_i - t_i \tag{3.5}$$

[2]Time interval that encompasses speech or music data rather than silence; transmission of silence is usually suppressed to save bandwidth.

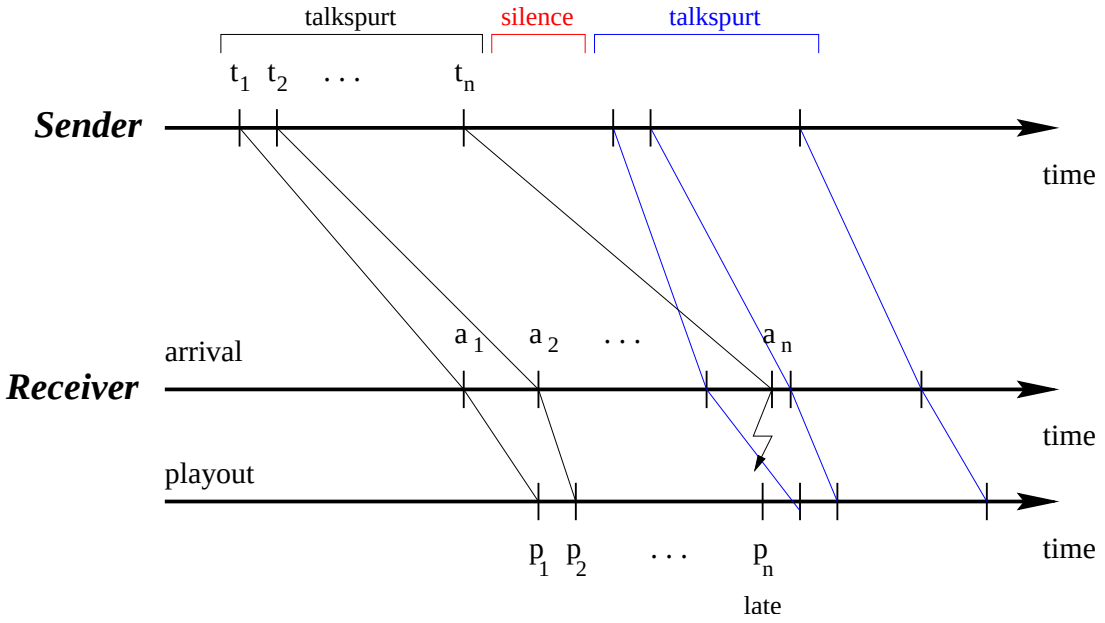


Figure 3.1: Timings associated with individual packets and their talkspurts

$$b_i = p_i - a_i \tag{3.6}$$
$$dp_i = d_i + b_i = p_i - t_i \tag{3.7}$$

The main problem of network delay estimation is in general that the sender and receiver clocks are not synchronized, and hence, it is not trivial to calculate the absolute delay. The delay variation required to estimate the playout delay, however, does not depend on absolute times. The jitter of the i-th packet can simply be calculated as follows:

$$j_i = |(a_i - a_{i-1}) - (t_i - t_{i-1})| \tag{3.8}$$

If packets would arrive in equal time intervals, meaning that packet jitter is zero, the audio packets could be immediately played back on reception. However, since packets on store-and-forward networks experience different transmission delays ($j_i > 0$), receiver buffering is an absolute necessity.

The calculation of the optimal playback delay has the competing goals of minimizing the extra delay introduced by buffering while maximizing the number of packets arriving prior to their playback time. Late packets that arrive after their playout point ($p_i < a_i$) are regarded as lost. Increasing the playout delay or in other words the buffering time, to prevent packets from being late, however, is not a good solution. Long playout delays to compensate for extreme delay variations increase the total end-to-end delays and thus, limit the usability of interactive real-time streaming applications.

3.1.4.1 Network Delay

In order to compensate for the jitter introduced by the network, receiver applications need to know the current delay and delay variation of the network. Several methods for network delay estimation ($D_{Network}$) have been proposed [Mon83, AC+93, R+94, MKT98]. Network delay estimation is calculated either statically at the beginning of a session, or dynamically by permanently adjusting the delay according to the instantaneous network state. Buffering mechanisms that rely on dynamical delay estimation adapt the buffering time to changing network delay variations.

Adaptation to delay changes in the network requires some form of filtering of past samples, such as a low-pass filter modeled after the TCP round-trip time estimator. For wide-area Internet transmissions the effects of sudden large changes in the delay, delay spikes, can skew network delay estimates badly. The study in [R+94] develops a network delay estimation algorithm that explicitly considers the phenomenon of delay spikes. Simulations based on wide-area Internet audio traces have proved that this estimator performs better than conventional buffering techniques without spike detection. A similar approach, used in the mechanism presented in [MKT98], is designed to recover quickly from sudden delay spikes and presents evidence of good performance.

An interesting relationship between buffering techniques simply controlled by the packet's transmission characteristic and open-loop error control schemes, such as FEC (see section 3.1.2), is documented in [D+94]. Generally, playout delay estimation based on the network jitter only does not provide adequate service when error control is also an issue. Supporting an open-loop error control scheme, such as FEC, requires modification of the receiver buffering algorithms. It is suggested that applications use sufficient buffering times to ensure with high probability that a copy of lost audio packets has arrived at the receiver before its playback point exceeds.

3.1.4.2 Processing Delay

Delay variations are not only introduced while the packet is transferred on the network. The receiving node, for example, introduces a so called processing delay when decoding, decompressing, mixing, etc. the audio data. Since normal user workstations are the end systems, process scheduling delay variations appear if non-real-time operating systems are used (see section 1.2.6). As a result, buffering to compensate jitter caused by irregularities in the process scheduling is required. These buffers, however, should not be controlled by user processes, because these processes clearly cannot compensate for scheduling delay variations. Therefore, audio devices usually provide special buffers for this purpose. Since scheduling delays usually do not vary greatly, these buffers can by fairly small.

The processing delay estimation ($D_{Processing}$) depends entirely on the end-node's operating system. The accuracy with which the scheduling delay variation can be measured depends

mainly on the timer accuracy, whereas the accuracy of the exact playout time of the scheduled packets depend on the operating system's scheduling granularity and, in particular, the minimum scheduling unit[3] of processes. Dynamic adjustment of the delay is preferable to static delay estimation, especially if the scheduling delay varies with different processing loads.

3.1.4.3 Summary

Summarizing one can conclude that receiver buffering, in order to compensate for the network delay variations and, less critically, to make up for the processing delays, plays an important role in media streaming application. In the context of real-time streaming, however, buffering delays should be as small as possible to minimize the total end-to-end delay and as big as necessary to accomplish the required loss characteristics. In general adaptive (dynamic) buffering delay estimations are preferable over simple (static) buffering mechanisms since "optimal" playout delay estimation depends highly on the network dynamics.

3.1.5 Summary

This section summarizes the analysis of application layer QoS mechanisms regarding their usability and importance for interactive media streaming. The following application layer techniques: packet transfer, forward error correction, adaptation and receiver buffering are examined.

With respect to packet transfer, interactive real-time streaming applications have to consider the following issues: First, for packet transfer RTP-on-UDP provides the best transport and streaming services among current Internet protocols. Second, the packet size used for media streams has the following tradeoff: it should be as small as necessary and as large as possible. Interactive streaming applications should use 1 at the most 2 media frames per packet to minimize packetization delay. Non-responsive streaming applications, in contrast, are recommended to use higher payloads in order to minimize packet overhead. Third, interactive real-time streaming applications are advised to "shape" their data traffic such that media packets are sent isochronously over time rather than in bursts. This has the potential to reduce packet loss rates since the likelihood of packet clustering is minimized.

Packet-based forward error correction mechanisms that are capable of correcting several consecutively lost packets provide good service for interactive real-time streaming in the Internet. Since the number of consecutively lost packets is usually small (in the order of 1

[3]Current operating systems have scheduling units of the order of 0.1 ms.

to 3 packets), these FEC mechanisms are effective within Internet communication. Packet-based FEC used in conjunction with traffic shaping, that isolates packet loss, improves the effectiveness.

The general technique of adaptation that adjusts the operation of an application depending on the QoS provided by the network is very effective within Internet real-time streaming. It has been shown that even quite severe service fluctuations can be accommodated by means of adaptive mechanisms. However, adaptation is limited to only compensate for QoS degradations of deterministic bounds. Since resource reservation mechanisms are not yet supported in most parts of the Internet, application level adaptation is absolutely necessary within real-time streaming.

Receiver buffering in order to compensate the network jitter and the jitter introduced by processing irregularities in the sending host is crucial for Internet communication if no hard QoS guarantees are granted. Adaptive buffering time estimation is beneficial since the network QoS characteristics change permanently in the Internet. Moreover, buffering within the sound device is necessary to compensate the jitter introduced by process scheduling within the receiver node. To minimize the buffering at the sound device, adaptive buffering mechanisms, that adjust their operation based on the measured scheduling jitter, are preferred. In general, one can conclude that adaptive (dynamic) buffering delay estimations are preferable over simple (static) buffering mechanisms since "optimal" buffering time estimation depends highly on the jitter dynamics.

3.2 Network Layer QoS

This section introduces various different network layer mechanisms to achieve network level QoS or differentiated QoS classes in the Internet. First, techniques that are commonly called service differentiation mechanisms are introduced; in particular, the IETF's Differentiated Service architecture is presented in section 3.2.3. Second, a network service that provides QoS guarantees based on resource reservation is explored. This network service or QoS framework is known as Integrated Services (see section 3.2.5).

3.2.1 Relative Priority Marking

Relative priority marking is a service differentiation mechanism that aims at providing QoS to real-time media streaming applications.

An examples of the relative priority marking model is IPv4 precedence marking [Pos81]. In this model the application, host or proxy node selects a relative priority or "precedence" for a packet (i.e., delay priority or discard priority), and the network nodes along the delivery path apply the appropriate priority forwarding behavior corresponding to the priority value within the packet's header.

The problem of these simple priority mechanisms is to prevent the application from marking all packets with the highest priority. For example, file transfer applications (such as FTP) that are not time critical compared with real-time applications could also improve their performance simply by marking packets as high-priority. In general most applications perform (much) better, if higher priorities are given to their data traffic.

In addition, since packets are marked individually without reference to their end-user nodes or applications, accounting for QoS provided to a certain customer cannot be done on shared networks within this model; semantics for end-to-end service, such as flows are lacking.

The Differentiated Service architecture, described in section 3.2.3, can be considered to be a refinement of this simple model. It emphasizes the role and importance of boundary nodes and traffic conditioners and introduces an enhanced per-hop behavior model that permits more general forwarding behaviors than relative delay or discard priority.

3.2.2 Service Marking

Service marking, which is a very similar idea to relative priority marking, is another mean to support some form of QoS. The IPv4 *Type of Service (ToS)* [Alm92] and the IPv6 *Traffic Class* are examples of a service marking model in the Internet.

Each packet is marked with the desired ToS. The ToS is defined by means of one or a set of the following service requests: "minimize delay", "maximize throughput", "maximize reliability" or "minimize cost". Network nodes are responsible to select routing paths or forwarding behaviors that are suitably engineered to satisfy the service request.

This service model is slightly different from the Differentiated Service (DiffServ) architecture (see section 3.2.3), because DiffServ does not use the ToS or traffic class field as an input for the routing decision. Also, the ToS markings, as defined in [Alm92], are very generic and do not span the range of possible service semantics. Service marking does not easily accommodate new services types (since the header field is small), and new types would involve changes in the configuration of "TOS → forwarding behavior" associations in each network node. Moreover, in service marking, requests can only be associated with individual packets whereas DiffServ also supports aggregate forwarding behavior for a sequence of packets. Another disadvantage of service marking over DiffServ is that it implies standardized services offered by all network providers. This, however, should be outside the scope of the IETF and left to the network providers themselves.

3.2.3 Differentiated Services

Differentiated Services (DiffServ or DS) [B+98], a scalable architecture for service differentiation in the Internet, is currently being discussed within the IETF.

This architecture achieves scalability by implementing complex classification and conditioning functions only at network boundary nodes, and by applying *Per-Hop Behaviors (PHB)*[4] to aggregates of traffic which have been appropriately marked using the DS field[5] in the IP headers. The difference to other service marking models is mainly that the service classes are not limited to a pre-defined standard, but are rather flexible.

Packets are classified and marked to receive a particular per-hop forwarding behavior on nodes along their path. Therefore, sophisticated classification, marking, policing and shaping operations need to be implemented at network boundaries or end-user hosts. Network resources are allocated to traffic streams by service provisioning policies that determine how traffic is marked and conditioned upon entry to a differentiated services-capable network, and how this traffic is forwarded within that network.

PHBs are defined to permit a reasonably granular means of allocating buffer and bandwidth resources at each node among competing traffic streams. As a result *per-application flow* or *per-customer* forwarding state, that limits scalability, does not need to be maintained within the core of the network. Service provisioning and traffic conditioning policies are sufficiently decoupled from the forwarding behaviors within the network interior to permit implementation of a wide variety of service behaviors, with room for future expansion.

3.2.3.1 Service Model

The differentiated services architecture is based on a simple model where the traffic entering a network is assigned to a service class. Each service class (or behavior aggregate) is identified by a single DS code-point[6]. Within the core of the network packets are forwarded according to the per-hop behavior associated with the DS code-point.

The key components of the differentiated service architecture are described here.

Domains and Regions A *Differentiated Service domain (DS domain)* is a contiguous set of DS nodes that operate with a common service provisioning policy and a set of PHB groups implemented on each node. DS domains have well-defined boundaries consisting of DS boundary nodes that classify and possibly condition ingress traffic to ensure that packets which transit the domain are marked appropriately to select a PHB from one of the PHB groups supported within the domain. Normally DS domains consist of one or more networks under the same administration; for example, an organization's Intranet or an ISP. The domain administration is responsible for ensuring that adequate resources

[4]The externally observable forwarding behavior applied at a DS-compliant node to a DS behavior aggregate.

[5]The ToS field of the IPv4 header or the Traffic Class of the IPv6 header when interpreted in conformance with the DiffServ architecture.

[6]A specific value of the DS field that is used to select a PHB.

are provisioned and/or reserved to support the domain's service levels. Since non-DS-compliant nodes within a DS domain may result in unpredictable performance and may impede the ability to satisfy the desired service levels, such nodes should be avoided within DS domains.

A *Differentiated Services region (DS region)* is a set of one or more contiguous DS domains. DS regions are capable of supporting differentiated services along paths that span the domains within the region. The DS domains within a DS region may support different internal PHB groups and different "code-point → PHB mappings". However, to permit services that span across the domains, the peering DS domains must establish a peering service level.

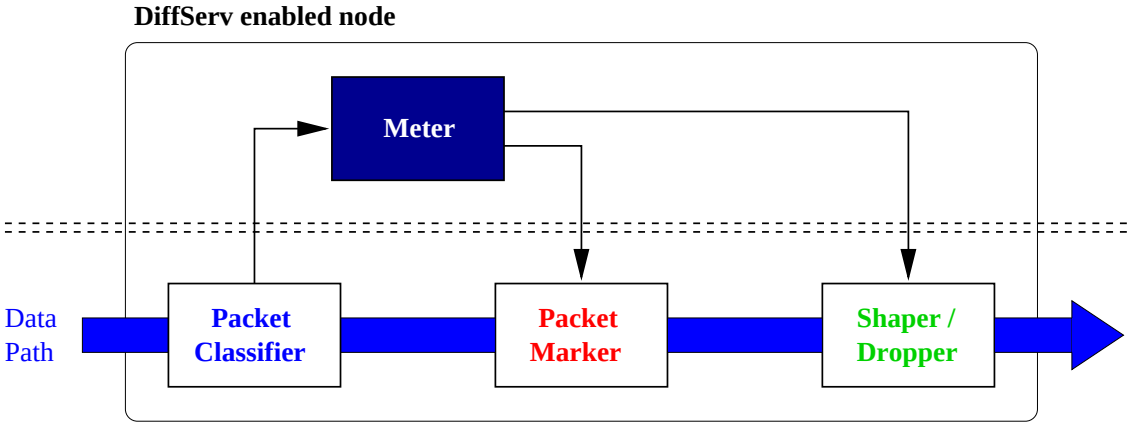


Figure 3.2: DiffServ Packet Classifier and Traffic Conditioner

Traffic Classification and Conditioning Differentiated services are extended across a DS domain boundary by establishing a *Service Level Agreement (SLA)*[7] between an upstream network and a downstream DS domain. Briefly, the SLA specifies packet classification and re-marking rules. The packet classification policy identifies the subset of traffic that may receive a differentiated service by being conditioned and/or mapped to a behavior aggregate. The inter-operation of the traffic classifier and conditioner is illustrated in Figure 3.2. A traffic stream is selected by a classifier that steers the packets to a logical instance of a traffic conditioner. A traffic conditioner may contain the following elements: meter, marker, shaper and dropper. A meter is used (where appropriate) to measure the traffic stream against a traffic profile. The state of the meter with respect to a particular packet (for example, whether it is in- or out-of-profile) may be used to affect a marking, dropping or shaping action. Thus, traffic conditioning performs metering, shaping, policing and/or re-marking to ensure that the traffic entering the DS domain conforms to the rules of the domain's service provisioning policy.

[7] A service contract between a customer and a (SLA) service provider that specifies the forwarding service a customer should receive.

3.2.3.2 Forwarding Behavior

The externally observable "forwarding behavior" applied to a particular service class of a DS node is described within the node's PHB. Per-hop behavior is defined in terms of behavior characteristics relevant to service provisioning policies rather than in terms of a particular implementation mechanism. PHBs can be specified by means of their resource priority relative to other PHBs or in terms of their relative observable traffic characteristics. Since PHBs are the means by which a node allocates resources to behavior aggregates, they should be specified in groups that share common constraints applied to every single PHB, such as a packet scheduling or buffer management policy. Multiple PHB groups may be implemented on a node and utilized within a domain in order to support a wide variety of different absolute and relative service classes.

3.2.3.3 Summary

DiffServ provides a simple QoS framework for the Internet without the stringent need for changes in end-user applications and end systems. Even if only parts of the end-to-end transmission path support DiffServ, the service operates well within these bounds. Unlike end-to-end resource reservation mechanisms such as IntServ/RSVP, DiffServ does not require that all network nodes along a delivery path support the service.

The DiffServ architecture has the potential to resolve the scalability problem of the IntServ architecture in the core of the network, since no "per-flow" state is required within network elements. In contrast, end-to-end QoS guarantees, as supported by IntServ, cannot be accomplished. The fact that DiffServ depends on the resource allocation mechanisms provided by "per-hop" behavior implementations, prevents DiffServ from offering end-to-end service guarantees. Moreover, the lack of a reliable admission control mechanism impedes DiffServ from offering reliable end-to-end resource promises. For example, in the case of a route change there is nothing to prevent links from being overloaded. The lack of admission control is the main reason why DiffServ is often deprecatingly considered to be a simple service providing only a "better" and a "worse" *best-effort* service.

3.2.4 IP Label Switching

In IP label switching [R+98a], path forwarding state and traffic management or QoS state are established for data streams on each hop along a network path. Traffic aggregates of varying granularity (i.e. packets, cells, flows) are marked with a forwarding label and associated with the corresponding label-switched path (or virtual circuit) at ingress nodes. IP switches perform the lookup of a packet's next-hop node, its per-hop forwarding behavior and the replacement label at each hop based on the current forwarding label.

The label switching model permits finer granularity resource allocation to traffic streams than, for example, service or priority marking, since label values are not globally significant

but are only meaningful on a single link. Therefore, resources can be reserved for the aggregate of packets or cells received on a link with a particular label. The label switching semantics allow traffic streams to follow a specially engineered path through the network simply based on the forwarding label. This finer granularity comes at the cost of additional management and configuration requirements to establish and maintain the label switched paths.

IP label switching is generally a very efficient approach, especially if media streams are long-lived (i.e. consist of many data packets). In this case, IP label switching is by far more efficient than regular IP routing due to the simpler decision making process in the network routers (or switches). If only few packets are transmitted (i.e. in the case of short-lived media streams), however, the cost of setting up the switching paths might not be worth. Since label switching can be processed very efficiently within network nodes, it has the potential to improve delay sensitive applications such as interactive audio tools by reducing the processing delays in every intermediate node.

The amount of forwarding state that must be maintained at each label switching node scales in proportion to the number of participating end-user nodes of the network in the best case (assuming multipoint-to-point label switched paths), and it scales in proportion to the square of the number of end-user nodes in the worst case, when end-to-end label switched paths with provisioned resources are employed.

As a result one can conclude that even though IP label switching is a very flexible and efficient approach to improve the QoS in the network, it does not scale very well in the core of the network, where IP switches have to cope with several thousand flows at a time.

3.2.5 Integrated Services

The IETF's *Internet Integrated Services (IIS or IntServ)* [Wro97b, Wro97a, SPG97] architecture provides network level QoS by controlling the network delivery service. This enables QoS sensitive applications to request their QoS desires by means of resource reservation mechanisms.

3.2.5.1 The Framework

The Integrated Services framework provides the ability for applications to choose different controlled levels of delivery service for their data packets. Before sending the data packets applications use a reservation mechanism to establish the resource reservation for the data stream. Different levels of QoS guarantees, namely *soft* and *hard* guarantees, are supported within this framework.

In order to support resource reservation within the Internet two entities are required. First, all network elements (IP routers) along the delivery path of an application's data

flow must support mechanisms to control and provide the QoS required by packets of the reserved flow. This is called the *QoS control service*. Second, a protocol to communicate the application's QoS requirements to the individual network elements along the path and to convey QoS management information between network elements and the application must be provided. This is referred to as *reservation setup mechanism*.

In the IntServ architecture the *QoS control* is provided by either the *controlled-load* or *guaranteed* service. Section 3.2.5.3 describes both QoS models in detail. The *Resource reSerVation Protocol (RSVP)* is currently the mechanism of choice within the Internet for reservation setup. Section 2.3.1 provides a detailed description of RSVP. Another resource reservation mechanism that supports reservations within IntServ is called YESSIR (see section 2.3.2 for details).

This memo emphasizes the clear line between the *QoS control services* provided by network routers and the *reservation setup mechanisms*. A common misunderstanding is that RSVP encompasses both functionalities. But, it is merely a protocol to communicate the QoS requirements to the network nodes. Furthermore, the interfaces to the *QoS control services* of IntServ are specified in a general manner, so that the services can be used in conjunction with different *reservation setup mechanisms*.

3.2.5.2 Reservation Setup Mechanism

The *reservation setup mechanism* is responsible for establishing and maintaining the resource reservation along the transmission path.

In order to invoke the *QoS control service* within the network elements, several types of data must be exchanged between the application and those network elements:

First, information generated at the sender, describing the data traffic (the sender *TSpec*) of the sender application is carried to intermediate network elements and to the receiver(s).

Second, information generated or modified within the network elements and required at the receivers to make reservation decisions, encompass the available services, delay and bandwidth estimates, and operating parameters used by specific *QoS control services*. The information is collected from network elements and carried towards receivers in so called *AdSpec* messages. Rather than carrying information from each intermediate node separately to the receivers, this information represents a summary, computed as it passes each individual hop along the transmission path.

Third, information generated by each receiver, describes (a) the QoS control service required for the reservation (guaranteed or controlled load), (b) a description of the traffic level for which resources should be reserved (the receiver *TSpec*), and (c) whatever parameters are required to invoke the service (the receiver *RSpec*). This information is carried from the receiver(s) to intermediate network elements, and finally, if the reservations have been installed successfully, to the sender in so called *FlowSpec* messages. The *FlowSpec*

describes the QoS parameters of a data flow, and if the resources are granted, it specifies the resource reservation.

In order to associate a *FlowSpec* with a particular data flow, a *FilterSpec* is required. The *FilterSpec* specifies the packets that can make use of the reserved resources (see Figure 3.3).

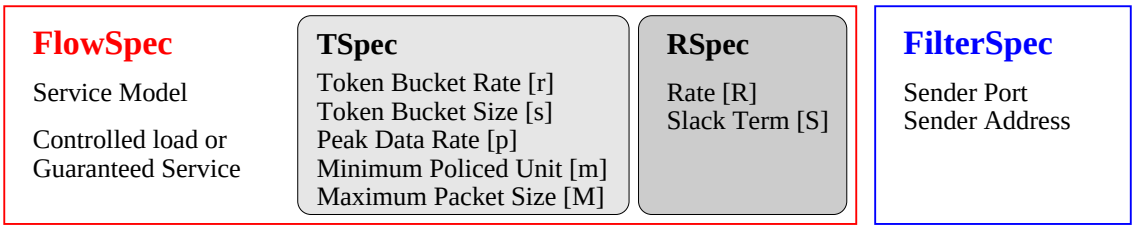


Figure 3.3: The IntServ Reservation Request Format: FlowSpec and FilterSpec

3.2.5.3 QoS Control Services

Service Models

Traditionally the Internet provides the same QoS to every data packet. This service is known as simple *best-effort* service (see section 1.2.4). The network promises no boundaries on delay, jitter or loss rates. It simply tries to deliver the packets as soon as possible.

Guaranteed Service

Guaranteed service [SPG97] provides firm bounds on end-to-end packet queuing delays. It offers sufficient service for real-time streaming application with hard QoS requirements. Guaranteed services make use of resource reservation mechanisms in order to provide fixed, guaranteed bounds on end-to-end delay and jitter. Guaranteed service does not explicitly minimize the jitter. It merely controls the maximum queuing delay and hence provides an upper bound for the jitter.

The concept behind guaranteed service is that a flow is described using a token bucket. Based on this flow description, network elements (routers, subnets, etc.) compute various parameters describing how the service element will handle the packets of this flow. By accumulating the parameters of all network elements along a transmission path, the maximum delay that a packet might experience can be determined.

Token Bucket Model A *token bucket* is determined by two parameters, a rate r and a depth b. An illustration of the token bucket model is shown in Figure 3.4. The bucket is continuously filled with tokens at rate r. It is limited to contain at the most b tokens. A *token bucket filter* is used to characterize a flow. A source that conforms to the token bucket filter (r, b) can only send when the bucket contains enough tokens. Thus, r can be interpreted as the long-term average rate of the flow, whereas b is its burst size.

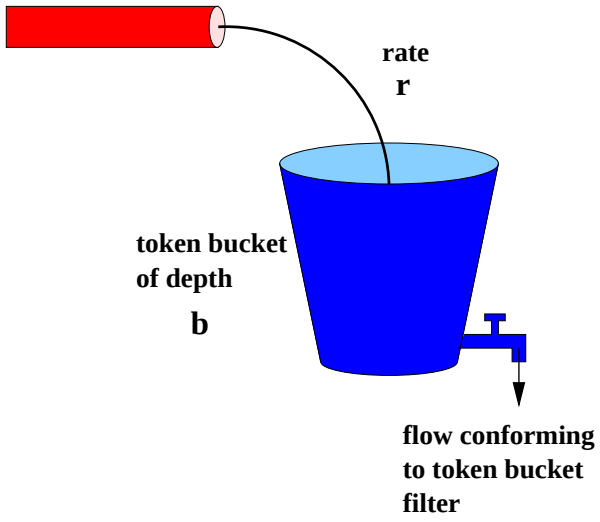


Figure 3.4: The Token Bucket Model

Guaranteed service relies on the traffic (*TSpec*) and the desired service (*RSpec*) of the flow reservation. The *TSpec* takes the form of a token bucket (r, b) plus a peak rate p, a minimum policed unit m and a maximum datagram size M. The *RSpec* is a rate R (with $R \geq r$) and a slack term S. The slack term signifies the difference between the desired delay and the delay obtained by using the reservation level R. It can be used by the network routers to reduce the QoS properties of their reservations for a particular flow.

The definition of guaranteed service is based on the results found by studying the *fluid model* [PG93]. It has been mathematically proven that the maximum queuing delay in a node is bounded by the time the last packet of a burst of size b is delayed in the queue, if guaranteed service provides a guaranteed bandwidth not less than r.

The bound on the queuing delay must be adjusted by error terms C and D in each network element which specify how the flow deviates from the fluid model. The composition function is applied along the entire path to compute the end-to-end sums of C and D, namely C_{tot} and D_{tot}. Then, the fixed latency of the path is added to get a bound on the absolute delay. Applications decide whether the resulting delay bound is sufficient based on their traffic characteristics and the received information on error terms and latency from the network.

In order to ensure that the traffic originating from the sender conforms to the token bucket, guaranteed service requires traffic policing at the edge of the network. According to the

specification [SPG97], non-conforming datagrams should be treated as normal *best-effort* datagrams. Besides standard policing, guaranteed service supports policing by means of *traffic reshaping*. Reshaping tries to restore (possibly distorted) traffic that violates the *TSpec* in such a way as to make it conform to the *TSpec*. It entails, for example, delaying packets until they are in conformance with the token bucket. Policing is done at the edge of the network, whereas reshaping is done in the core of the network (at all source branch points and at all source merge points).

Guaranteed services can be seen as one extreme end of delay control for networks. Other services controlling delays, usually provide much weaker assurances about the resulting delays. In order to provide absolute assurance of QoS, guaranteed services must be provided by every network element along the transmission path. This, however, does not imply that guaranteed services must be deployed in the whole Internet. If fully deployed in an Intranet or the backbone of an ISP, guaranteed services can be offered within the company's Intranet or between customers of the ISP.

Controlled Load Service

Controlled load service [Wro97a] promises a QoS service for data flows that is closely approximating the QoS that the same flow would receive from an unloaded network with *best-effort* service. In contrast to guaranteed service it does not give explicit or hard QoS guarantees with respect to bandwidth, delay and jitter. Nonetheless, the service provides low end-to-end delays and very few packet losses similar to those packets would experience under unloaded conditions from the same series of network elements. Note, the term "unloaded" is used in the sense of "not heavily loaded" or "not congested" rather than not loaded at all.

Applications that exploit controlled load service for their data flows may assume: first, the percentage of packets not successfully delivered to the receiving end-nodes is very small (close to the basic packet error rate of the transmission medium); second, the transit delay of most delivered packets is not greatly exceeding the minimum transmit delay experienced by any successfully delivered packet.

Controlled load service is designed for applications that are highly sensitive to overloaded conditions (for example, live audio/video streaming tools) but can tolerate small QoS variations (for example, adaptive applications). Applications with fixed end-to-end timing requirements that fail immediately when end-to-end delays exceed their boundaries require hard QoS guarantees (see guaranteed service 3.2.5.3). According to the discussion in section 3.1.3, adaptive real-time streaming applications operate badly under overloaded or congested network conditions. But, network experiments [M+98] have shown that they work well on unloaded networks. Thus, a service that imitates unloaded networks might provide sufficient QoS for these applications.

The controlled load service is similar to the guaranteed service in that sender nodes communicate the token bucket specification (*TSpec*) of their traffic (see guaranteed service

3.2.5.3) to the network. The network nodes ensure that enough resources will be set aside for the data flow. Active admission control mechanisms prevent the network elements from overloading their link, buffer space or computational capacity of their forwarding engine. As in the case of guaranteed service, controlled load service provides QoS only for traffic conforming to the *TSpec* given at setup time. The token bucket parameters require that the amount of data sent for a particular flow does not exceed $rT + b$, where r and b are the token bucket parameters and T is the length of the time period. Excess traffic should simply be forwarded on a *best-effort* basis if sufficient resources are available. Network elements should prevent excess controlled load traffic from unfairly impacting the handling of normal *best-effort* traffic. The proper handling of the latter case, however, is not yet clearly defined. Controlled load service also allows delaying of packets for reshaping reasons. However, the overall requirement for limiting the duration of such traffic distortion must be considered.

Controlled load service is advantageous over guaranteed service in that it enables much better overall network utilization. Note, guaranteed resource reservations often waste a large percentage of the available resources since they are not always fully used. Hence, controlled load service offers a more cost-effective solution if no real hard guarantees are required. Finally, since controlled load service provides only soft QoS guarantees, the implementation of the router software is less complex and therefore easier to realize than in the case of guaranteed service.

Traffic Control

The IntServ *QoS control service* within Internet nodes is implemented by mechanisms that are collectively called *traffic control*. These mechanisms include *packet classification*, *admission control*, and *packet scheduling* or some other link-layer-dependent mechanism to determine when individual packets must be forwarded. *Admission control* is required to ensure the node has sufficient resources available to meet the QoS demands of each new request. The *packet classifier* determines the QoS requirements for each packet coming through the node based on its flow affiliation. For each interface, the *packet scheduler* or other link-layer-dependent mechanisms are responsible for providing the promised QoS. See Figure 2.9 as an illustration of the inter-operation between those components.

The implementation issues of *traffic control* are largely left to the router vendors. Current implementations differ in the processing algorithms, underlying mechanisms, processing speed, and bandwidth utilization. The IntServ working group has suggested one possible implementation framework [B$^+$94]. The scheduling algorithm *Weighted Fair Queuing (WFQ)* forms a key part of this framework. WFQ supports fair scheduling based on multiple queues. Each flow has a separate queue and packets are scheduled so that each flow receives a fair fraction of the link bandwidth depending on its priority. In contrast to static time-slicing mechanisms, WFQ does not waste bandwidth while there is still demand. If a particular flow does not use its assigned bandwidth, other classes can use it. It dynami-

cally determines the next queue to be served according to the bandwidth that each queue received previously and its priority. Other scheduling algorithms, such as weighted round robin[8], could also be used within IntServ implementations. However, since WFQ has a much finer granularity of fairness, it is usually the preferred scheduling algorithm within the Integrated Services architecture despite its computational complexity.

The traffic control approach suggested if IntServ and normal *best-effort* service operate side-by-side within the network is illustrated in Figure 3.5. It shows how the network bandwidth is shared in a hierarchical model where guaranteed service is at the top level, followed by the less strict controlled load service and finally the simple *best effort* service.

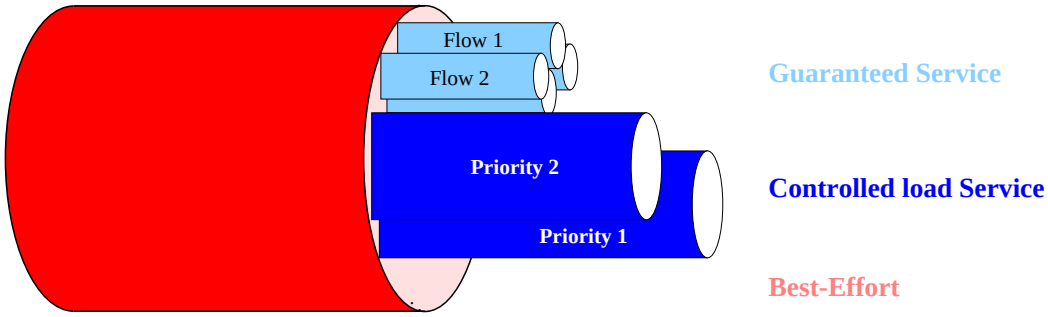


Figure 3.5: Hierarchical Traffic Control

At the top level each guaranteed service flow is assigned to a separate WFQ queue. As a result all guaranteed service flows are strictly separated from each other and scheduled in a fair manner with respect to their bandwidth requirements. All other traffic (controlled load and *best-effort*) is assigned to a pseudo WFQ flow. Within this flow controlled load traffic and best-effort traffic is differentiated by means of priorities [Sch97]. In order to prevent *best-effort* from being totally blocked the network admits only a certain amount of controlled load traffic. The controlled load service class is divided into subclasses with different delay bounds by assigning different priorities. In order to utilize the bandwidth assigned to controlled load service efficiently (for example, during bursty periods), higher priority classes are allowed to temporarily borrow bandwidth from a lower-priority class. Within each controlled load subclasses the overall delay is minimized simply by a FIFO scheduling algorithm. Flows within a subclass should have similar QoS characteristics, so when a burst occurs in one flow, the other flow can share the delay without being too much delayed.

[8]Weighted round robin provides fair service in terms of the number of packets sent by each flow and the flow's priority.

3.2.5.4 Summary

IntServ can be seen as a reliable QoS framework capable of providing soft and hard end-to-end QoS guarantees to end-user applications.

The main drawbacks, however, are: First, IntServ relies on all network elements along a transmission path supporting the service. Although IntServ is the main QoS architecture (apart from DiffServ) currently experimented within the Internet, hardly any network routers are already capable of supporting IntServ. Second, end-to-end QoS guarantees require a per-flow state in every intermediate network element. This, of course, does not scale in large networks and in particular not within the core of the network.

Due to the scalability problem of IntServ researchers have begun to discuss new, scalable QoS models for the core of the Internet. The DiffServ architecture, for example, is one result of this effort. Furthermore, aggregation mechanisms which make IntServ more scalable in the core of the network are explored [Ber98, GA98].

A clear trend towards IntServ or DiffServ cannot be seen yet and might never be seen. New approaches or approaches that incorporate the IntServ and DiffServ architectures are more likely to become the future QoS framework within the Internet. One such integrated approach is presented in section 3.2.6. Since many believe that the future QoS solution for the Internet includes the ideas of DiffServ and/or the IntServ, industry leading companies investigate both approaches.

3.2.6 Integration of Differentiated and Integrated Services

The Integrated Services architecture (see section 3.2.5) supports end-to-end QoS with soft or hard guarantees on IP networks. However, the reliance on "per-flow state" and "per-flow processing" is an impediment to this deployment in the Internet at large, and especially in large backbone networks. Differentiated Services (see section 3.2.3), on the other hand, promise to expedite the realization of QoS enabled networks by significantly simpler mechanisms compared to IntServ without requiring end-to-end deployment. DiffServ overcomes the implicit scalability problems of IntServ and is therefore suitable for large networks, such as the Internet. In contrast to IntServ, however, DiffServ provides a significantly weaker QoS model without QoS guarantees.

The deployment of DiffServ in the core network (where scalability is a concern) and IntServ in stub networks at the edges (where scalability is not crucial) meets the requirements for a scalable global QoS architecture for the Internet. Since IntServ and DiffServ are complementary tools in the pursuit of QoS, a framework that uses IntServ as "customer" of DiffServ has chances for success [Ber98].

3.2.6.1 Network Architecture

A sample network shown in Figure 3.6 is illustrated in order to demonstrate inter-operation between IntServ and DiffServ.

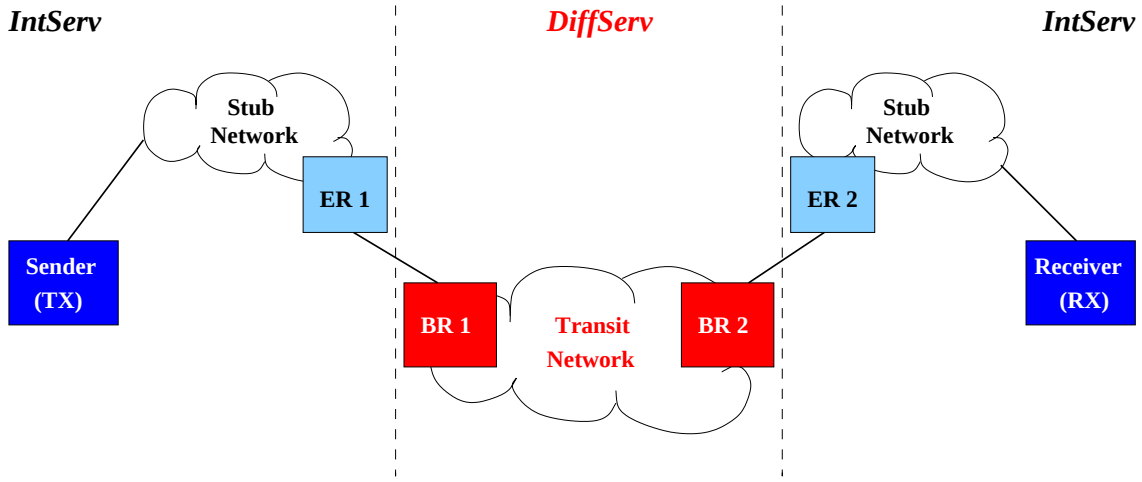


Figure 3.6: A sample Network Configuration: DiffServ capable transit network and two IntServ capable stub networks

The transmitting (TX) and receiving (RX) hosts use a *resource setup mechanism*, such as RSVP (see section 2.3.1), to communicate the QoS requirements of QoS aware user applications to the network nodes. Both, TX and RX, are part of IntServ stub networks. The transit network, on the other hand, is not required to be IntServ capable. It provides DiffServ with two or more levels of service based on the DS field in the headers of carried packets. In order to provide end-to-end QoS services, the transit network must be able to carry messages of the *resource setup mechanism* transparently to other stub networks.

The IntServ service types (controlled load and guaranteed service) must be mapped to a DiffServ service class (or behavior aggregate). End-to-end services can be provided by concatenating PHBs [B+98] (see section 3.2.3). The contract negotiated between the customer (owner of the stub network) and the carrier (owner of the transit network) for the capacity to be provided by each of a number of standard DiffServ service classes is called the carrier-customer agreement.

Edge routers (ERs) are special routers that bridge the IntServ and DiffServ region of the network. End-to-end resource reservation requires that the applications and the IntServ nodes are explicitly informed of admission control failure in the DiffServ network. This enables them to take corrective action and to avoid overloading the DiffServ network. The DiffServ implementation within ERs is responsible to provide an interface to the *DiffServ Admission Control Service (DACS)*. If dynamic service agreements between stub networks and transit networks are deployed, the DACS must communicate with so called *bandwidth*

brokers [N+97] to make admission control decisions based on provisioned limits as well as on the topology and the capacity of the DiffServ network. Boundary routers (BR) that are simply conventional DS boundary routers are expected to implement the policing function of DiffServ ingress routers.

3.2.6.2 Reservation Establishment

Any network element involved in reservation establishment must reject a reservation request if insufficient resources are available in order to prevent already reserved flows from losing their QoS guarantees. Since the transit network does not interpret the reservation setup messages, ERs[9] are responsible to control admission. They must compare the "requested" resources with the resources "available" at the corresponding DiffServ service level in the transit network. If the reservation is admitted, the DACS must update the available capacity for the service class and propagate it to the bandwidth brokers of the DiffServ network.

3.2.6.3 Summary

The inter-operation of IntServ and DiffServ to provide a scalable QoS framework for today's Internet is successful in that it resolves the scalability problem of IntServ in core networks. However, the hard end-to-end QoS guarantees that IntServ can offer become lost by mapping IntServ reservations onto DiffServ service classes. It is important to note that DiffServ does not offer end-to-end services. Even though the framework described in this section approximates the end-to-end QoS of IntServ fairly well, the lack of a reliable admission control mechanism for DiffServ prevents the service from offering reliable resource promises and results in getting overloaded under certain conditions.

3.2.7 Summary

This section summarizes the study of network layer QoS mechanisms that are currently of interest within Internet research. The QoS mechanisms are examined on their usability and importance for real-time media streaming.

The first group of mechanisms includes various service differentiation mechanisms, namely relative priority marking, service marking, and DiffServ.

The simple approach of relative priority marking is only of limited use for real-time streaming since relative priorities do not provide suitable differentiation. A major flaw of this approach is also that nothing prevents applications from marking all packets with the highest priority.

[9] If RSVP is used, the ER that receives the RESV message after it passes the transit network.

Service marking enhances service differentiation based on relative priorities by increasing the range of possible service semantics. However, service marking has no provision to easily add new service types since the header field is small and new types would involve changes in each network node. Unlike DiffServ, service marking uses the marking also as an input for the routing decision.

DiffServ is currently the primary differentiation mechanism discussed within the IETF. It outperforms the two former differentiation mechanism by supporting flexible service classes that are not limited to a pre-defined standard. In principle, DiffServ provides a mechanism that divides the network into several virtual *best-effort* networks each of that offers different QoS. Since DiffServ does not require "per-flow" state information within network routers, it has the potential to resolve the scalability problem of IntServ in the core of the network. However, forwarding behaviors only on a "per-hop" basis, prevent DiffServ from offering end-to-end service guarantees. Moreover, the lack of a reliable admission control mechanism impedes DiffServ from offering reliable resource promises. Even though DiffServ cannot guarantee end-to-end QoS, it has the potential to improve network QoS received by real-time streaming application when widely deployed in the Internet. QoS sensitive real-time media traffic would then be protected from *(discrete) data traffic*.

Another mechanism, called IP label switching, aims at improving current QoS in the Internet by means of packet switching techniques. IP label switching is more efficient than regular IP routing due to the simpler decision making process in network routers (or switches). This has the potential to reduce the overall processing load on the network and the end-to-end delays received by real-time streams simply by reducing the processing cost in every intermediate node. Similar to IntServ, IP label switching does not scale in the core of the network, where IP switches have to maintain forwarding state on a "per-flow" basis.

The IntServ architecture provides network level QoS by controlling the network delivery service. Real-time streaming applications can request their QoS demands by means of a resource reservation protocol. Granted QoS provides optimal service for QoS sensitive applications such as real-time streaming applications. IntServ supports *guaranteed* (hard) and *controlled load* (soft) QoS guarantees. On the one hand, *controlled load* QoS, providing service equivalent to unloaded networks, is suitable for adaptive real-time streaming applications that are capable of dealing with small variations in the QoS. On the other hand, *guaranteed* QoS, offering hard QoS guarantees, provides optimal service for real-time streaming applications even without adaptation, error correction, and receiver buffering mechanisms. The main drawbacks of IntServ can by summarized as follows: first, IntServ relies on all network elements along a transmission path to support end-to-end reservations, and second, "per-flow" state is required in every intermediate network element. This, of course, does not scale in large networks and in particular not within the core of the Internet.

Another approach that integrates IntServ and DiffServ suggests to use DiffServ as a scalable, hop-by-hop QoS mechanism in the core of the network, and IntServ at the stub networks, where scalability is not a problem. Using IntServ as "customer" of DiffServ en-

ables streaming application to negotiate their QoS requirements within stub networks. In the DiffServ network in the core, IntServ QoS reservations must be mapped to appropriate DiffServ service classes. Since the DiffServ cloud in the core cannot easily provide real QoS guarantees, this integrated approach cannot offer hard end-to-end QoS guarantees such as pure IntServ networks.

3.3 Summary

This section summarizes the results of the analysis of application level techniques and network level QoS mechanisms which impact the QoS of interactive real-time media streaming applications.

The conclusions of the discussion on application layer techniques can be summarized as follows:

- With respect to packet transfer, interactive real-time streaming applications should consider that:
  - RTP-on-UDP provides the best streaming service for interactive media streaming applications among current Internet protocols.
  - Interactive applications demand small packets that encompass only 1 or 2 media frames.
  - Real-time streaming applications need to "shape" their data traffic
- Packet-based forward error correction mechanisms capable of correcting a few consecutive packet losses are recommended.
- Receiver buffering is mandatory within Internet communication where no hard QoS guarantees are available.
- Adaptation as a general mechanism for adjusting the operation of an application depending on the received QoS is highly recommended. It has the potential to improve most QoS mechanisms (for example, receiver buffering, forward error correction, etc.).

From the study of network layer techniques the following results can be concluded:

- The IETF's DiffServ and IntServ architectures both have the potential to improve the network QoS for real-time media streaming applications.
- IP label switching, which has the potential to significantly reduce the end-to-end delay experienced by packets and the processing cost in every intermediate network node, does not scale successfully in the core.

- Since DiffServ cannot provide QoS guarantees, IntServ with its hard and soft QoS guarantees is more profitable for real-time media streaming.
- The scalability problems of IntServ prevent its use in the core of the Internet.
- Integrated solutions where DiffServ is used as a scalable, hop-by-hop QoS mechanism in the core of the network, and IntServ is used at the stub networks, are more likely to become the future QoS framework of the Internet.
- Since DiffServ in the core cannot easily provide real QoS guarantees, integrated solutions cannot offer hard end-to-end QoS guarantees. Therefore, extended IntServ solutions that achieve scalability through aggregation and overhead reduction might be preferred in due course.
- IntServ and IP label switching are technologies that inter-operate well and therefore might direct future Internet networks away from classical packet routing.

Chapter 4

The Application: WebAudio

This chapter describes the application architecture and discusses the implementation issues of WebAudio, the real-time audio streaming application developed within the context of this thesis.

Before introducing the application the results of the discussion on Internet multimedia protocols (see chapter 2) and a summary of Internet real-time streaming issues are presented.

Summary on Internet Real-Time Audio Streaming

Interactive live audio communication tools stream real-time audio data; this data is highly delay sensitive (see section 1.3). Streaming audio can thus be classified as *time-critical traffic* (see section 1.1) with very high QoS constraints in terms of the end-to-end delay, jitter and reliability.

Short round-trip delays are particularly important since real-time conferencing, like human communication in general, demands good interactivity. Even small amounts of jitter introduced by the network and the end systems is noticeable since it has a direct impact on the latency introduced by receiver buffering and thus on the overall end-to-end delay. Besides delay and jitter, real-time media streaming is also very sensitive to unreliable data transfer. In Internet communication, only packet loss is normally considered since bit errors occurs very rarely and usually result in packet discard. The bandwidth requirements of audio streaming are moderate compared with the average throughput of current Internet applications; compared with the bandwidth requirements of packet video they are rather negligible. In addition, sophisticated encoding formats are proposed which greatly reduce the data rate of audio streaming to as little as 2.4, 4.8, 13 or 16 kbps for voice and approximately 128, 196, or 384 kbps for high quality sound (see section 1.3).

Since end-to-end QoS constraints are very strict in the case of interactive real-time audio, QoS degradations have a great impact on usability (in terms of user satisfaction). In order

to satisfy the QoS requirements of audio streaming applications, the IntServ architecture (see section 3.2.5), capable of supporting hard QoS guarantees [1], is currently the most promising approach in the Internet. If applications are capable of adapting to small variations in the QoS, soft guarantees in IntServ[2] would provide satisfactory service. According to the discussed in section 3.1.3, it has been shown that adaptive applications perform well under low or moderate network load. If IntServ is not supported on every node along the delivery path, end-to-end reservations cannot be established. In this case applications should resort the DiffServ architecture (see section 3.2.3). DiffServ operates even if only few segments of the network support service differentiation. Since DiffServ does not promise end-to-end QoS, applications must be able to adapt to variations in network QoS.

If neither IntServ nor DiffServ is provided by the network segments between a sender and receiver, applications must rely solely on adaptation mechanisms to compensate for the dynamic QoS changes in the network. Since both IntServ and DiffServ are still in the experimental stage and are not widely deployed within today's Internet, adaptation is highly recommended for real-time streaming applications. In the wide area Internet, QoS variations often exceed the adaptive boundaries of adaptation mechanisms, and hence, satisfactory usability can rarely be accomplished. This situation is unacceptable and will hopefully change as the latest generation of routers with IntServ and DiffServ QoS support become more widely deployed.

Real-time audio streaming applications intended for use within the global Internet cannot rely on the availability of network level QoS. However, if an application can benefit from any QoS support from the network, it should make use of it. One can conclude that packet audio, in the context of Internet streaming, demands the "best" QoS that the network can currently offer. Therefore, applications should have support for both IntServ and DiffServ, and should be capable of adapting to the "best" available service. If no network QoS support is provided, applications must use adaptation mechanisms to compensate for the heavily changing QoS characteristics of the network. The impact of variable end-to-end delays due to network congestion can be reduced by dynamic receiver buffering (see section 3.1.4). High packet loss can be largely resolved by means of open loop forward error correction mechanisms such as packet-based FEC (see section 3.1.2) which is an effective approach to compensate for isolated packet loss. Packet losses can be isolated by means of traffic shaping by spreading packet clusters (see section 3.1.1). If the available network bandwidth is insufficient, applications should provide adaptive encoding mechanisms that allow dynamic changes of encoding format. Thus, if congestion occurs, applications can decrease the throughput requirements by applying higher compressed audio encodings.

[1] Guaranteed services promise resources or QoS with only few exceptions; a route change, for example, may cause the network to lose reservations (see section 2.3.3 for a detailed discussion).

[2] Also known as controlled load service (see section 3.2.5.3).

Motivations for "another" Audio Application

It could be argued that developing another streaming application was unnecessary since there are already several commercial applications (such as, Microsoft NetMeeting, RealPlayer, CuSeeMe) and even applications with free source code (for example, Berkeley's vat [MJ95], UCL's rat [H+95], INRIA's FreePhone [BVGFPne], etc.) available for the Internet. The main reasons for developing a new application from the ground up are as follows:

- An important goal was to analyze the benefits of the new Internet protocol for QoS based real-time audio streaming. As it is shown in section 5.3, IPv6 has the potential to improve current QoS mechanisms. Since none of the currently available audio streaming tools (see section 1) had support for IPv6 at the time this work was started, the only option was to develop a new tool.
- Developing a new application from ground up allows proper integration of new protocols and mechanisms from the beginning, and also simplifies modifications and testing.
- Current Internet multimedia applications are mainly limited to either adaptation or resource reservation and are not capable of adapting between different QoS mechanisms, namely IntServ and DiffServ, and simple "best-effort" service.
- Today's audio streaming applications are usually controlled through their individually developed user interfaces rather than Web based interfaces. With the growing use of the Web as a means to provide selection and control interfaces, users now expect state-of-the-art Internet applications to provide Web based interfaces. Current audio applications tend to have been developed prior to this rapid growth of the web and are often difficult to integrate into Web based environments.

The rest of the chapter is organized as follows. Section 4.1 focuses on the architecture of the new real-time audio streaming application. The design issues are discussed on a conceptual level. Section 4.2 discusses the implementation issues and the problems occurred during the development.

4.1 Application Architecture

A brief overview of the new real-time audio streaming tool is provided by highlighting the main characteristics of WebAudio.

Client-server based: The application is based on client-server architecture. The audio client is called *wa*, whereas the audio server is called *was*.

Asymmetric architecture: The client and the server are divided into two separate applications. Thus, interactive communication requires an audio client and server on both ends.

Simplex communication: Audio information is transmitted via a simplex transport channel. Hence, two-way communication requires two separate channels – one in each direction.

Network level QoS support: WebAudio makes use of resource reservation, if supported by the underlying network, to "guarantee" the QoS required for the audio streams. If the network does not provide QoS, the streams are transmitted assuming the simple *best-effort* service.

Adaptation support: The first release of WebAudio only supports adaptation of receiver buffering. However, WebAudio has provision for the future addition of adaptive mechanisms such as dynamic encoding selection or adaptive packet-based FEC due to the built-in QoS feedback channel from the client to the server.

Out-of-band signalling: WebAudio uses a special signalling connection, independent from the media streaming channel, to initialize and control the stream. Separating stream control from the media channel enables the use of common protocols rather than developing specialized and non-compatible protocols.

Multi-streaming capable: Both, the WebAudio client and server, are capable of managing multiple streams simultaneously. The server sends the encoded audio data to all the clients on point-to-point connections. The clients mix all received data streams prior to playback.

Web interface: WebAudio is specifically designed to be a Web application. The "open" control interface allows simple remote control of the applications by means of a Web browsers, and as a result, stream control is easily integrated within Web pages or Web applications.

4.1.1 Operational Overview

This section provides an overview of (1) how the *WebAudio server (was)*, the *WebAudio client (wa)* and the user interfaces operate, (2) what forms of communication are required, (3) which protocols are used, and (4) how these components work together. Figure 4.1 provides an operational overview of the WebAudio application framework and illustrates a typical WebAudio scenario of a stream setup and teardown.

The WebAudio server listens on a "well-known" control port for HTTP-based or RTSP-based control requests from clients. In the Web scenario illustrated in Figure 4.1, the Web browser requests an initial control Web page or a user interface from the WebAudio server

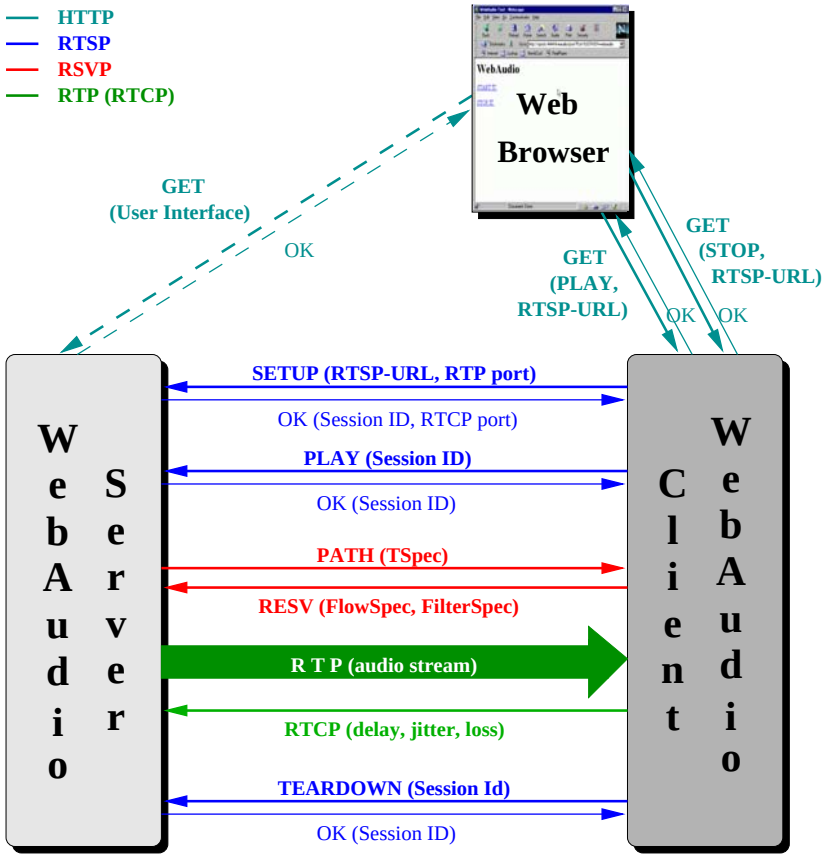


Figure 4.1: Operational Overview of the WebAudio Framework

by a standard HTTP request. The Web page prompts the Web browser to startup the WebAudio client as a so-called *helper application* (see section 4.2.3).

The user interface, which could be a simple HTML page or a Java Script enhanced Web page, provides stream controls based on hyper links to remote control the WebAudio client (see section 4.1.5). Although the client provides a HTTP-based and an RTSP-based control interface, in this scenario only the HTTP interface is used.

Audio streaming is initialized by a HTTP request from the user interface to the WebAudio client. The request includes the WebAudio command `PLAY` and an RTSP-URL, namely the address and port of the server and the required audio resource (see section 2.4.2). The client transforms this request into an RTSP `SETUP` request by adding the RTP port where the audio stream is expected and forwards it to the WebAudio server (see section 4.1.4.2). The server sets up a new audio stream, and if successful, the server indicates the successful operation in the RTSP reply. The reply message also contains a session identifier for the new audio session and the server's RTCP port where the QoS feedback information of the client is expected. Upon successful stream setup, the WebAudio client requests the server to start streaming the audio data by sending an RTSP `PLAY` request. The WebAudio server

then starts streaming the audio to the client RTP port.

Upon receiving the audio stream packets, the client estimates the optimal buffering time in order to compensate the jitter experienced in the network and calculates the playout time of the packets (see section 4.2.5). At the predetermined playout time, the audio frames of the packets are decoded and mixed (if several streams are received) and finally played back (see section 4.2.8). The client computes a receiver statistic which includes QoS parameters such as delay, jitter and packet loss based on the current network QoS characteristics. This feedback information is periodically sent back to the WebAudio server by means of RTCP (see section 4.1.4.3).

The audio stream is torn down by a HTTP-based `STOP` request from the user interface. The WebAudio client again transforms the control request into an RTSP `TEARDOWN` request and forwards it to the WebAudio server. The server then tears down the stream and releases the audio session state.

If the underlying network supports resource reservation based on RSVP, the server requests the server RSVP daemon to send `PATH` messages indicating the required resources for the audio stream towards the WebAudio client. If RSVP is supported along the whole transmission path, the client receives the `PATH` messages and then establishes the reservation by requesting the client RSVP daemon to send `RESV` messages up-stream (see section 4.1.3.2). At the end of an audio session, the reservation is torn down by the client by releasing the RSVP session. The RSVP daemon then sends `TEARDOWN` messages up-stream to the sender which releases the reservations in the network nodes.

4.1.2 Architecture

The WebAudio system is based on an asymmetric client-server architecture. The basic architecture of the WebAudio system, the server *was* and the client *wa*, is illustrated in Figure 4.2. Even though the WebAudio client and server have conceptually different functionalities, they have many tasks in common. The modular application architecture, therefore, enables reuse of modules in both applications.

The communication and networking interface of the WebAudio server is very similar to the client interface. Both have TCP modules (see section 2.2.2) to exchange application control information. The WebAudio server has one TCP module listening on a well-known port for control requests of the client. The client, in contrast, requires two TCP modules: one to listen on a well-known port for incoming requests from the user interface, and one to communicate with the server.

For the stream control both have a HTTP and an RTSP module (see section 2.4) which process the stream control messages exchanged between the user interface, the client and the server. The client and server TCP modules which receive the control requests of either the user interface or the client are connected to the HTTP and RTSP modules. They are

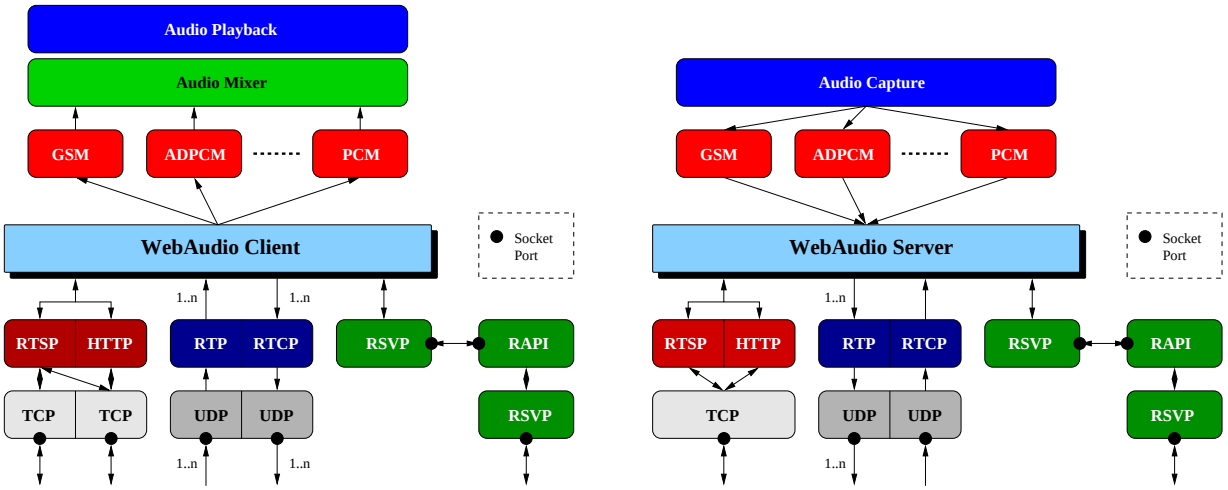


Figure 4.2: Modular Architecture of the WebAudio Client *wa* and Server *was*

capable of processing either HTTP or RTSP requests. The second client TCP module, which is used to for the client-server communication, supports only RTSP.

As a transport layer mechanism for the actual audio data stream, UDP (see section 2.2.1) is used. The application level protocol RTP with its feedback mechanism RTCP (see section 2.2.3) is used to facilitate streaming. Both the WebAudio client and server require an RTP, an RTCP and two UDP modules. The WebAudio server listens on one UDP port for the RTCP feedback messages. On the other port, it sends the RTP audio packets to the client(s) receiving the stream. The WebAudio client, in contrast, listens on one UDP port for the RTP audio packets, whereas the other port is used to send the RTCP feedback messages.

The QoS support of the first version of WebAudio is limited to the IntServ framework (see section 3.2.5). In order to establish and maintain resource reservations, RSVP is used as a resource reservation protocol (see section 2.3.1). Both the client and the server have an RSVP module to interface with the *RSVP Application Programming Interface (RAPI)* and the RSVP daemon running in the end systems. In later releases support for the DiffServ QoS architecture (see section 3.2.3) will be added.

While the communication modules of the WebAudio server are very similar to those of the client, the audio processing is different. The server uses a module to capture the audio data from the sound device and several encoding modules for the different audio formats. The WebAudio client, in contrast, has different modules for each decoder, a module for the mixing of multiple streams received simultaneously, and a module to playback the sound samples.

The main modules of both applications, shown in the center of the illustrations (see Figure 4.2), interconnect the different communication and audio modules. The client and server modules are responsible for calling the modules in an appropriate sequence to ensure that

the data are processed according to their time constraints. The WebAudio applications are based on a single threaded program flow. All modules are implemented such that no blocking system calls lock the process in a module. Based on the BSD "select"[3] and polling[4] mechanisms asynchronous processing is achieved.

The individual modules introduced as part of this application architecture overview are further examined and discussed in latter sections of this chapter.

4.1.3 Protocols

The WebAudio application framework requires several protocols on different OSI levels to communicate stream control information for the application signalling and to transmit the audio data between the server and clients.

4.1.3.1 Network Level

On the network level WebAudio supports both, the current Internet protocol as well as the new Internet Protocol, IPv6 (see section 2.1).

Since one aim of this work is to explore the benefits of IPv6 for network level QoS mechanisms, including support for IPv6 was essential. As shown in section 5.3, the WebAudio implementation is used to examine the gain of IPv6 and, in particular, the IPv6 flow label within packet classification. Since IPv6 still is not wide-spread within the Internet, the development of a state-of-the-art audio streaming application intended for use in today's global Internet, however, requires also support for IPv4.

To support both network protocols in an integrated fashion, the transport protocol classes (or modules) for TCP and UDP are developed such that they can be configured to use either IPv4 or IPv6. The application is currently designed such that the network protocol need to be specified during start up. One might have expected that the underlying network protocol is pre-determined by the operating system release. However, current IP implementations with support for IPv6 still provide service for IPv4. Both protocols operate side-by-side in a so-called dual-stack architecture.

IPv6 support within WebAudio is limited to audio streaming communication. Since WebAudio is intended to be easily integrated into Web applications, the application signalling needs to be based on top of the network protocol used by normal Web clients. Standard Web browsers, such as Netscape's Communicator 4.x and Microsoft's Internet Explorer

[3] A technique which indicates buffers, sockets, or files which either have more data in their input buffers to read or free space in their output buffers to write further data.

[4] A mechanism which periodically checks if further processing is required, or if new data are available to be read or written.

4.x, however, do not support IPv6 yet[5]. Hence, WebAudio control signalling is based on top of IPv4. This is not really a limitation, since the analysis of the IPv6 benefits for QoS mechanisms concerns only the time-critical data traffic in the first place. The low-bandwidth and discrete data traffic of the control signalling is not much affected by IPv6 and thus can simply be transmit via IPv4.

4.1.3.2 Resource Reservation

WebAudio has support for network QoS based on resource reservation as described in the IntServ framework (see section 3.2.5). The resource reservation mechanism known as RSVP (see section 2.3.1) is currently the reservation protocol of choice in the context of IntServ.

Since IntServ has the potential to improve the network QoS for QoS sensitive or time-critical data traffic my far, WebAudio provides support for this QoS mechanism and resource reservation based on RSVP. Resource reservation within WebAudio, however, is limited to the audio data traffic. Stream control messages are transmitted based on the simple *best effort* approach of IP. Since this traffic is not time-critical, except that it might delay the setup and teardown of streams slightly, it does not discriminate the application.

The inter-operation between the WebAudio client and server applications and the RSVP process is illustrated in Figure 2.9. The server or client application communicates with the RSVP daemon of the local machine by means of the RAPI. Library functions provided by RAPI to setup and teardown reservations facilitate the communication between the user application and the RSVP process.

The reservation establishment process described here is illustrated in Figure 4.3. The WebAudio client and server register a new session with the local RSVP daemon by calling the *rapi_session* function during session initialization. The destination IP address and transport protocol port of the audio receiver uniquely identify the new RSVP session. The applications also need to specify an asynchronous call back function which is used for upcalls on RSVP events such as reservation error, new path message, reservation established, etc. In order to establish a new reservation, the WebAudio server calls the *rapi_sender* function by passing the QoS parameters of the audio stream. The local RSVP daemon then establishes the `PATH` state for the new flow and initiates the sending of the `PATH` messages. When the first `PATH` message reaches the client host, the RSVP daemon informs the client application about the `PATH` event by means of the call back function. The client then calls the *rapi_reserve* function. The receiver needs to specify the QoS parameters for the `RESV` messages. If the reservation is established, the sender RSVP daemon upcalls the WebAudio server with a reservation established event.

A reservation teardown is processed similarly. The server or client application unregisters

[5]Recently Microsoft has released a beta IPv6 upgrade for the Internet Explorer.

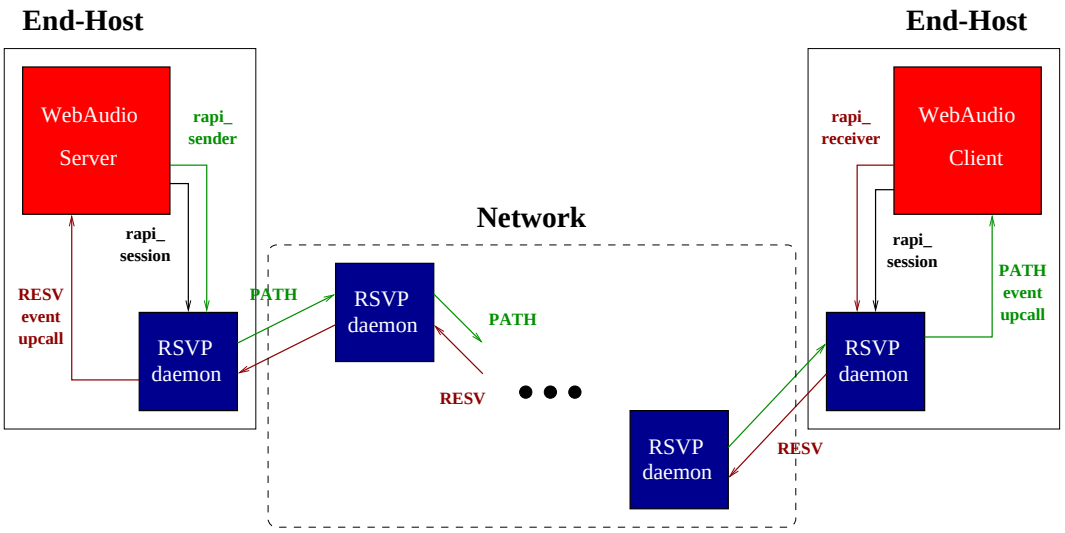


Figure 4.3: Reservation Establishment with RAPI

the RSVP session with the RAPI function *rapi_release*. As a result, the local RSVP daemon creates a `TEARDOWN` message and sends it on the reservation path.

The interest in the impact of the IPv6 flow label for packet classification demanded the extension of the RSVP protocol and the RSVP daemon. The protocol changes are documented in an official Internet-Draft [SDRS98]. In order to support the IPv6 flow label in the RSVP software, the RSVP daemon implementation (rel 4.2a3) from ISI [ISI98] was extended. It was the only RSVP implementation with a freely available source code at the time. This implementation is also the most wide-spread RSVP distribution and served as reference implementation for many subsequent RSVP products. The results of the experiments using the IPv6 flow label as packet classification criterion rather than the source and destination ports of the transport protocol header are presented in section 5.3.

In summary, WebAudio has built-in support for standard and flow label based RSVP. The RSVP mode to be used is specified during application initialization.

4.1.3.3 Transport Level

The transport protocols used within WebAudio are discussed with consideration of the different traffic types.

First, the transport protocol requirements for *stream control traffic* are examined. WebAudio deploys either HTTP or RTSP as stream control protocols. HTTP, on the one hand, is limited to be used with TCP (see section 2.4.1). RTSP, on the other hand, can be used either over TCP or UDP (see section 2.4.2). Since TCP provides ideal service for the transport of control information due to its reliable service properties, it was the transport

protocol chosen for both, HTTP and RTSP based stream control. Limiting the transport protocol for *stream control traffic* to TCP reduces the complexity of supporting multiple protocols for the same end and allows the processing of HTTP and RTSP messages through a single interface.

Second, the demands on the transport protocols for *audio data traffic* are discussed. The choice of transport protocols within IP networks is currently limited to either UDP or TCP (see section 2.2). Based on the results of the comparison of these transport protocols (see section 2.2.4), UDP was selected as the transport protocol. However, since RTP facilitates media streaming, the *audio data traffic* is transfered based on RTP-on-UDP. The main advantages of RTP-on-UDP as opposed to TCP or simple UDP for media transfer can by summarized as follows (compare with Table 2.2):

- RTP-on-UDP allows applications to fully control the data rate. Transport level rate control, such as in TCP, does not reduce the data rate required be the application.
- RTP provides useful media streaming information, such as the timestamp, sequence number and media type, in the protocol header.
- RTP's feedback mechanism provided by its control protocol RTCP is highly valuable when adaptation mechanisms are deployed.

The use of RTP within WebAudio can be described as follows: The WebAudio server wraps the audio data in RTP packets and adds media streaming information, such as the payload type, sequence number, timestamp and the session ID, in the protocol header. The clients use these information to calculate the network QoS characteristics, namely the delay, jitter and packet loss, of the audio stream. The QoS characteristics are fed back periodically to the WebAudio server in so called *Receiver Reports (RR)* (see section 4.1.4.3). The RRs are attached to an RTCP header and then sent in UDP packets to the server's RTCP port. The feedback information include the last sequence number, the loss fraction, the total loss, the jitter, and the delay. Although the QoS feedback accomplished by means of RTP and RTCP has provisions for server-side adaptation, the current version of WebAudio does not yet implement adaptive mechanisms at the server.

4.1.3.4 Application Support Level

WebAudio requires an application support layer protocol that facilitates *stream control* of the audio streaming. The *Real-Time Streaming Protocol (RTSP)*, which recently became a "Proposed Standard" [S+98b] within the IETF, provides the basic control functionality for real-time media applications. It provides control services similar to a VCR remote control with operations like play, pause, and stop.

Besides several advantages of RTSP, which have already been mentioned in section 2.4.3, the main reasons for deploying it within WebAudio are noted here:

- RTSP is a well defined standard. This allows for inter-operation with third-party applications that use RTSP as stream control protocol (for example, the RealPlayer media tool suite[6]).
- RTSP is specifically designed for the purpose of stream control. Its control functionality is therefore conform with the need of real-time streaming applications.
- RTSP's protocol syntax is very similar to HTTP. Thus, RTSP has the potential to easily inter-operate with HTTP.

Even though RTSP seems to be the perfect solution for stream control, it has one small disadvantage for WebAudio. The integration of WebAudio in Web environments requires a user interface – an application or any other control mechanism – which enables the control of WebAudio from within Web pages. The stream control protocol RTSP, however, demands for sophisticated user interface applications, such as Java programs, ActiveX controls, Web browser plug-ins or helper applications which implement the RTSP protocol. Simple HTML or Java Script based user interfaces would not be sufficient since these simple techniques do not support RTSP.

However, these simple approaches support the Web protocol HTTP. As a result, HTTP is supported as an alternative stream control protocol to RTSP. The advantages of the HTTP based control interface are obvious:

- HTTP support allows easy Web integration of WebAudio. A simple HTML based Web page can be used as a user interface. HTTP control requests can be encoded in normal hyper links. A click of such a control link causes the Web browser to send the respective HTTP control request to the client application.
- HTTP support enables the use of standard Web browsers as very simple user interfaces. Control requests can simply be entered in the Web browser's *location field*[7] (see Figure 4.5).

The multi-protocol control request interfaces is implemented such that HTTP and RTSP stream control requests can be processed (see section 4.2.9). As a result, the WebAudio client and server can be controlled by either HTTP or RTSP. However, to facilitate communication between the client and the server, only the more sophisticated stream control protocol is used.

4.1.4 Application Interface

The application interfaces of WebAudio can be described from two different perspectives: stream control and audio streaming. The interfaces are described with respect to the supported protocols: HTTP, RTSP, and RTP (RTCP).

[6]Further information are available at http://www.real.com/.

[7]The field where users usually enter the URL of a new Web page.

4.1.4.1 HTTP based Stream Control

The HTTP based stream control makes use of the HTTP searchpart mechanism (see section 2.4.1.1). It enables HTTP requests to be extended by a list of parameters. The mechanism allows the encoding of control commands, here called methods, and their corresponding parameters within standard HTTP-URLs.

The syntax of the HTTP-URL format used within WebAudio can be described as follows:

```
http://<host>[:<port>]/<resource>[?<method>{&<parameter>]}
```

The individual variables have the following meaning:

`<host>` specifies the Internet domain name (or IP address) of the WebAudio application.

`<port>` determines the port number of the application listening for incoming requests. This port is usually well-known or must be carried towards the client or user interface by some other means.

`<resource>` references a specific resource, usually a specific audio stream, to which the `<method>` should be applied. It may consist of a path and/or a file name (for example, `/liveaudio/gsm`).

`<method>` specifies the method which is applied to the `<resource>`. Three methods are currently supported: `GET`, `PLAY` and `STOP`.

`<parameters>` or the list of parameters determine method specific arguments of the control request. The supported parameters depend on the individual methods.

A brief description of the supported methods and their parameters is provided below:

`GET`: The WebAudio server simply replies the content of the specified `<resource>` as HTML page. If no method is specified in the HTTP request, `GET` is used as the default method. This allows WebAudio to be used as a Web server. If the `<resource>` string is empty, a default Web page which starts up the standard user interface and the WebAudio client application at the receiver is sent. The `GET` method does not support parameters.

`PLAY`: Additional parameters of the `PLAY` method determine either the WebAudio server (parameter: `SERVER`) to which the client should forward the control request or the RTP receiver port (parameter: `PORT`) where the client expects to receive the audio stream.

The following examples are presented for clarification: The first example presents a user interface requests from the WebAudio client (usually on the same machine, but it can be any Internet host) to set up and start streaming the audio resource `liveaudio/pcm` from the WebAudio server `webaudio` listening at port `4445`:

```
http://localhost:4444/liveaudio/pcm?PLAY&SERVER=webaudio:4445
```

Upon receipt of this stream control request, the client transforms the control request in an RTSP `SETUP` and `PLAY` control request forwards them to the server `webaudio`. The second example shows a HTTP `PLAY` control request directly send to the WebAudio server `webaudio.lancs.ac.uk`:

```
http://webaudio.lancs.ac.uk:4445/liveaudio/pcm?PLAY&PORT=8312
```

The server will then setup a new stream and start streaming the audio data to the source IP address of the HTTP request and the transport port `8312` of the HTTP URL.

`STOP`: This method causes the WebAudio client and server to teardown the specified stream and to release all the resources associated with that stream. `STOP` does not support any additional parameters.

In the case of unsuccessful request processing, WebAudio returns an error message with an error response code indicating the reason. Table 4.1lists the HTTP response codes used within WebAudio along with their meaning in this context. The general meaning of the response code classes of HTTP is presented in Table 2.7.

Code	Description
200	HTTP control request was successfully processed
403	No permission to access the audio resource identified in HTTP-URL
404	HTTP-URL or audio resource is not valid or not available
500	WebAudio application error
501	WebAudio command unknown
505	HTTP protocol or version not supported

Table 4.1: HTTP Response Codes used within WebAudio

From this description on how HTTP based stream control is achieved within WebAudio, one might conclude that the simple HTTP based approach is sufficient for the purpose of stream control. However, HTTP based stream control has two serious flaws.

First, standard HTTP does not provide a way for the WebAudio server to respond with an appropriate error message or error code in the case of a server error. WebAudio specific errors or errors related to stream control cannot be expressed by the default HTTP error codes and messages. This problem could be solved by extending the HTTP protocol. However, modifying the stream control protocol so that it is no longer compliant with HTTP removes the main advantage of HTTP based streaming control, and thus, standard Web browsers could not be used as simple user interfaces anymore.

Second, HTTP has no provision to exchange server generated session or stream identifiers to uniquely address individual media sessions or streams. The stateless information retrieval protocol HTTP does not provide semantics for media sessions or streams, and hence, the HTTP standard does not define a protocol header field to express such identifiers. As a solution the WebAudio server could transmit the session id as part of the HTTP payload. This solution, however, requires special HTTP clients that interpret the payload correspondingly. As a result, the solution was not considered. Another approach to address individual streams within HTTP stream control requests is to use the request source IP address in conjunction with the media resource specified in the HTTP-URL. It is to be noted that the source port information is not useful if stream control is carried out by means of Web browsers, because subsequent control requests of the same session might have different source ports. This technique limits the service to address one stream per destination address and resource within WebAudio. This of course is not a strong restriction, since common end systems usually have only one sound device. Receiving the same audio stream multiple times would not be of interest. To resolve this limitation, a simple solution could be to add the RTP port of of the WebAudio client as `PORT` parameter to the control request. It could then be used in concatenation with the source IP address to uniquely identify the client application.

4.1.4.2 RTSP based stream control

The general stream control mechanisms of RTSP are already described in detail in section 2.4.2. WebAudio implements the standard RTSP protocol according to RFC 2326 [S+98b]. In order to demonstrate how RTSP is used within WebAudio, all the RTSP methods used are described. Their application within WebAudio is shown in examples.

WebAudio implements only the mandatory RTSP functionality. The following methods are supported:

`OPTIONS` simply request a list of all the methods that are supported by WebAudio.

`SETUP` is called to initialize a stream. The stream state is allocated and a session number is generated. `SETUP` must be called before a stream can be played. An example of a SETUP request-response pair is presented below. It is assumed that the stream setup could be processed successfully.

```
SETUP rtsp://audioserver.lancs.ac.uk/liveaudio/gsm RTSP/1.0
CSeq:  302
Transport:  RTP, unicast, client_port=4588

RTSP/1.0 200 OK
CSeq:  302
Transport:  RTP, unicast, server_port=6256
Session:  234303923
```

Within the `SETUP` request, the client defines the transport protocol, the receiving transport port, and the communication approach (unicast or multicast). In the response, the server indicates its RTCP `server_port`. In order to allow the client to reference the stream in subsequent requests, the server generates a random and unique `Session` identifier which is also sent as part of the RTSP response. The `CSeq` parameter is used as a sequence number to uniquely identify request-response pairs. It is used in all RTSP requests or responses.

`PLAY` causes the WebAudio server to start streaming the audio data of the stream corresponding to the `Session`. `PLAY` simply changes the internal state of the stream from initialized to play. The following example shows how the `Session` parameter is used:

```
PLAY rtsp://audioserver.lancs.ac.uk/liveaudio/gsm RTSP/1.0
CSeq:  303
Session:  234303923
```

If the RTSP request could be processed successfully, the WebAudio server responses as follows:

```
RTSP/1.0 200 OK
CSeq:  303
```

`PAUSE` commands the WebAudio server to stop streaming the audio data. The stream or session state, however, is not released yet. A subsequent `PLAY` request starts the streaming process again.

`TEARDOWN` stops the audio streaming and releases the stream or session state in the server. A subsequent control request with this session id would fail since the stream does not exist anymore.

Table 4.2 lists the RTSP response codes that are used within WebAudio.

4.1.4.3 Data Streaming using RTP

For the transfer of the audio data, WebAudio uses RTP-on-UDP. Section 2.2.4 provides an in-depth discussion on the benefits of the different transport protocols, namely UDP and TCP, for time-critical data transmission such as audio streaming and the advantages of the stream protocol RTP. The protocol header introduced by the application layer protocol RTP merely adds stream information (see section 2.2.3) in front of the audio data. Figure 2.8 illustrates how the audio data are encapsulated in RTP, UDP and in the IP packet.

The stream information introduced by RTP (see Figure 2.7) is particularly valuable in the context of real-time streaming. The *timestamp* field allows the receiver to calculate the

Code	Description
200	RTSP request is processed successfully
400	RTSP method is invalid
403	No permission to access the audio resource
404	RTSP-URL or audio resource is not valid or not available
454	Session identifier is unknown or missing (stream might not be properly setup yet)
500	RTSP server error
501	RTSP method is not supported
505	RTSP protocol or version not supported

Table 4.2: RTSP Response Codes used within WebAudio

transit delay and the jitter of the received packets. It provides the means to calculate the optimal buffering time and the playback time of the packet. The *sequence number* field enables the client to restore the original packet order. Packets traveling along different transmission paths as a result of route changes might break the correct ordering. It is to be noted that UDP provides neither a service which guarantees the proper order of packets nor a mechanism to detect reordered packets. The *payload type* field indicates the audio encoding of the packet data. Even if the sender changes the audio encoding in the middle of a session (for example, due to temporary congestion), the change of the audio encoding is indicated, and hence, the client can properly decode the audio data of the packet. The *source identifier* field indicates the audio stream of the packet. It enables the WebAudio client to distinguish audio data of multiple streams received at the same time. The source identifier is required to correctly mix several streams received at the same transport port. It is especially valuable within multicast communication. The remainder of the fields, for example the flags, the version, etc. are merely required for protocol signalling reasons.

The stream information of the RTP header provide the necessary data to calculate QoS feedback for WebAudio servers. The current transit delay and jitter, the total packet loss and the loss fraction as well as the last and highest sequence number of received packets are returned within the RTCP receiver reports [S+96] (see section 2.2.3). The QoS feedback allows the WebAudio server to adapt the application operation in response to the changing QoS characteristics in the network.

The transit delay $delay_n$ and jitter $jitter_n$ of packet n are calculated as follows:

$$delay_n = arrivaltime_n - timestamp_n \tag{4.1}$$

$$jitter_n = jitter_{n-1} + \left\{ \frac{1}{16} \times (|delay_n - delay_{n-1}| - jitter_{n-1}) \right\} \tag{4.2}$$

The loss fraction of the report interval is determined as follows:

$$lossfraction = \frac{packets_{expected} - packets_{received}}{packets_{expected}} \tag{4.3}$$

RTCP is designed as a general control mechanism for RTP which includes, apart from RRs, also *Sender Reports (SRs)* and *Source DEScription (SDES)* reports. RTCP also supports aggregation of multiple reports within one RTCP packet. Within WebAudio, however, RTCP is only used for the purpose of providing a QoS feedback channel from the client to the server. WebAudio clients periodically send single RRs for each stream. Aggregation of multiple RRs is in most cases not suitable, since clients usually receive their audio streams from different WebAudio servers. The frequency of RR messages depends on the bandwidth used for the audio traffic. The feedback period is selected such that approximately 5% of the bandwidth utilized by the data traffic of the RTP stream is used for the RTCP traffic.

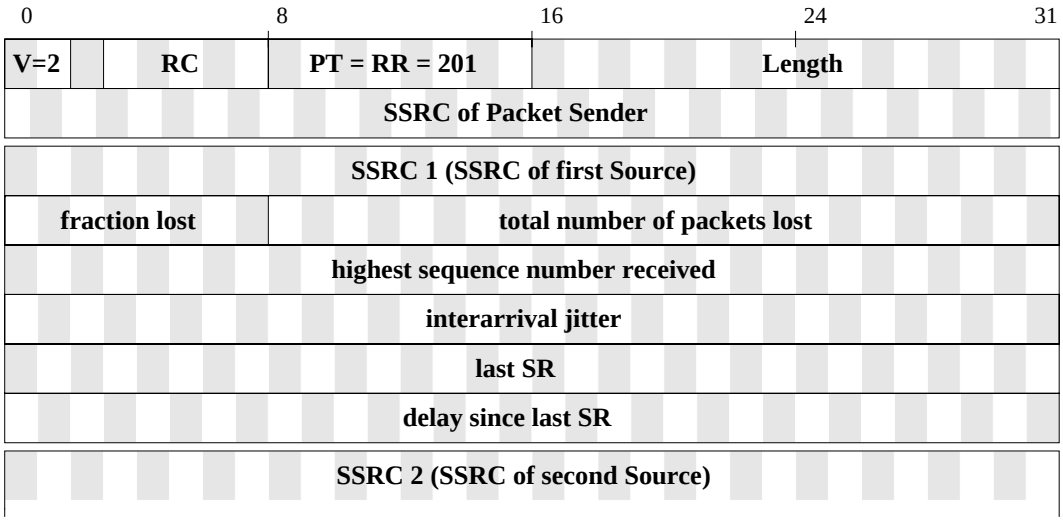


Figure 4.4: The RTCP Header Format

The message format of the RTCP receiver reports used within WebAudio is presented in Figure 4.4. For the transport of the RTCP messages, the simple datagram protocol UDP is used. The server's RTCP port is negotiated during stream setup by means of RTSP (see section 4.1.4.2). The RTCP feedback is an optional feature within WebAudio. If the WebAudio server does not receive feedback information from the client, it simply continues operation without QoS adaptation.

4.1.5 User Interface

WebAudio offers an "open" application control interface which enables a user interface to control the client and server applications either via HTTP or RTSP.

The simplest form of a user interface compromises a standard Web browser. In addition to retrieving Web pages, the browser is used to send HTTP based control requests to the WebAudio client. Control requests can be entered at the browser's *location field*. As specified in section 4.1.4.1, control requests can be simply encoded in HTTP-URLs. Figure 4.5 illustrates the usage of Netscape's browser as a basic user interface.

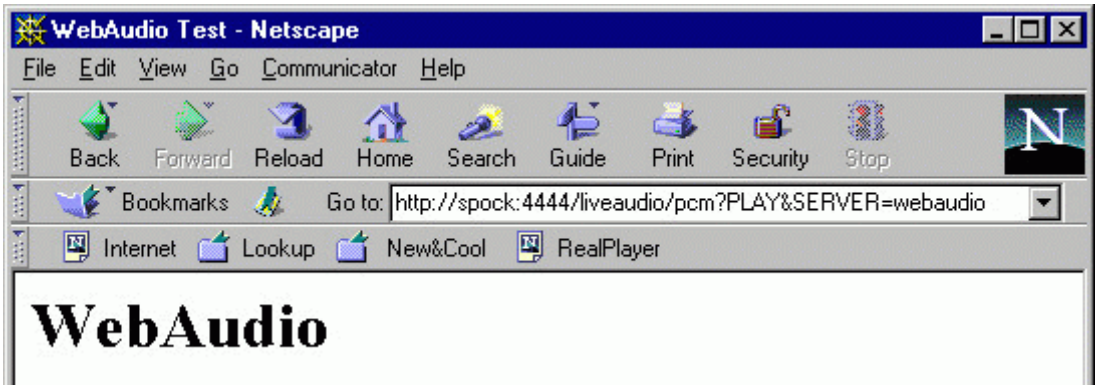


Figure 4.5: Netscape's Web Browser as a Simple User Interface

HTTP-URLs encoding the stream control commands can also be included in standard hyper links of HTML documents. The user can then launch the control commands by clicking on the links. It is advisable to use HTML frames to redirect the HTTP response of WebAudio to another frame in order to prevent the responses from replacing the "control" HTML page. The client responses indicate the success or failure of the commands. See section 2.4.1 for the complete list of HTTP response codes used within WebAudio. The WebAudio response might also include HTML content which is then displayed within the browser.

The following example shows how WebAudio control commands can be encoded in plain HTML. The "control" Web page could be used as a very simple user interface.

```
<HTML>
...
<A HREF="http://localhost:4444/liveaudio/pcm?PLAY\&
        SERVER=webaudio.lancs.ac.uk">
PLAY Live Audio (from webaudio.lancs.ac.uk)</A>
<A HREF="http://localhost:4444/liveaudio/pcm?STOP\&
        SERVER=webaudio.lancs.ac.uk">
```

```
STOP Live Audio </A>
...
</HTML>
```

A user interface programmed in Java Script [Co.pt], the Web script language, is also limited to control the WebAudio client by means of the HTTP interface. The script language is based on an event driven execution model. Programmers add functionality to the Web page by assigning Java Script functions to events, such as mouse pressed, mouse moved, load page, close page, etc. In the context of WebAudio, Java Script has the potential to provide enough functionality to design easy-to-use and attractive user interfaces. The next example shows how WebAudio control requests, encoded in HTTP-URLs, can be used within Java Script:

```
...
<SCRIPT>
function play () {
    waWindow =
    window.open('http://localhost:4444/liveaudio/pcm?PLAY\&SERVER=was',
                '','width=50,height=18');
    waWindow.write("<HR><P>Play</P><HR>");
}
function stop () {
    waWindow.load('http://localhost:4444/liveaudio/pcm?STOP\&SERVER=was',
                  10);
    waWindow.close();
}
</SCRIPT>

<BODY ONLOAD="play()" ONUNLOAD="stop()">
...
```

The `play()` and `stop()` functions shown in this example are interpreted upon a `LOAD` and `UNLOAD` event. The `play()` function opens a small window, sends the control request encoded in the HTTP-URL to the WebAudio client and writes the word "Play" in the new browser window. The `stop()` function simply sends a HTTP `STOP` control request to the client and closes the window.

User interfaces programmed in Java or any other general programming language is not limited to use HTTP for stream control. These sophisticated programming languages are capable of carrying out generic TCP communication. User interfaces based on programming languages without support for HTTP connections[8] need to implement either HTTP

[8]Java Script or Java, for example, have built-in support to open HTTP connections to send/receive HTTP requests/responses.

or RTSP as stream control protocol. Extensional programming APIs or libraries providing the basic protocol functionality of HTTP and RTSP significantly simplify the implementation of user interfaces. The advantages of using more sophisticated programming languages are, firstly, that RTSP, which has shown to be the more sophisticated stream control protocol, can be used, and secondly, that the application has full control of the protocol engine itself. The processing of the received responses messages is done entirely within the application. In the case of Web browser based user interfaces, the treatment of the HTTP responses is left to the Web browser.

If the user interface is implemented as a *Java Applet* [Co.va], the same reasoning can be applied. In this case, however, the limitations of *Java Applets* need to be considered. Applets are processed like Java Scripts within the context of a Web page. The *byte code*[9] of the Java program is not sent as part of the Web page. Only a reference to the *Java Applet* (in form of URLs) is included in the Web pages. In the context of WebAudio *Java Applets* have mainly one disadvantage compared to general Java applications. They can only open socket connections to the Internet host from which they were loaded[10]. Therefore, unless the *Java Applet* is loaded from the same host where the WebAudio client is executed (normally the local host), the Applet cannot establish a connection to this machine (even if it is the local host). This, however, is not a problem within WebAudio, since *Java Applet* user interfaces can easily be served by the WebAudio client. The HTTP protocol support of the WebAudio client, provides sufficient functionality to serve Web pages or Java byte code.

Summarizing, one can conclude that the "open" application control interface of WebAudio offers great flexibility to user interfaces. It enables, for example, simple integration of WebAudio into existing applications. Supporting the Web protocol HTTP as an application interface protocol has the advantage of easy Web integration of user interfaces (based on HTML, Java Script or Java). In addition, it enables the use of standard Web browsers as a basic user interface. The real-time streaming protocol RTSP is supported to enable sophisticated streaming control. Programming languages such as Java, C, C++, Pascal, etc., but not Java Script, allow the implementation of user interfaces with comprehensive stream control capabilities based on RTSP.

4.1.6 Scalability Considerations

Scalability is an important characteristic of group communication tools and conferencing applications. Since human communication often takes place in groups, the question how well an audio application scales in the number of participants is important.

The question of scalability has to be considered independently for sending and receiving applications. The reason for this is that sending audio information to many listeners is

[9] Java programs are compiled to *byte code* which is interpreted by the Java *Virtual Machine (VM)* during execution time.

[10] The Internet host which served the Applet byte code.

a different issue than listening to audio information of many senders. For the former procedure many every-day applications such as radio broadcasting, lecturing, etc. are known. In contrast, for the latter procedure hardly any applications where people listen to a large number of audio sources at the same time are known. Even human face-to-face conversation suffers greatly if several people talk simultaneously. As a result one can conclude that the scalability issue regarding the sending application must be differentiated form the receiving application.

In order to use WebAudio as a "broadcast" communication tool, the server must be able to support simultaneously a large number of receivers for the same flow. Since the first version of WebAudio is limited to point-to-point or unicast communication between the server and the client, the server application does not scale in the number of receivers. The reasons for not considering point-to-multipoint or multicast communication (see section 2.1.1.3) are: First, multicast is currently only available on the virtual multicast network known as the MBone. As discussed earlier, the performance of the MBone suffers greatly due to the fact that multicasting is achieved by means of IP tunneling. Moreover, the MBone is not widely deployed, and therefore it enables only a small portion of Internet users to benefit from multicast support. Second, the additional implementation work to support multicast would simple have exceeded the scope of this work. However, future releases of WebAudio will consider multicast support.

The WebAudio server is designed to provide service from small to moderate groups. Scalability depends mainly on the number of different encodings used, since the server has to process the costly operation of encoding the audio frames for each different audio format. Sending the packets several times (for each receiver) is computationally less expensive, but might be limited by the available bandwidth. The relationship between the cost of encoding and sending a stream is expressed in equation 4.4. The number of receivers is expressed by the value N whereas the number of different encodings used at the same time is determined by K. K is bound by the number of different codecs supported. The equation provides a simplified processing cost estimate for the server.

$$O_{Server}(N) = \sum_{k=1}^{K} O(Encoding[k]) + N \times O(Sending) \tag{4.4}$$

From this discussion one can conclude that due to the lack of multicast support the current version of WebAudio does not support broadcast to large groups. However, small or moderate groups (of the order of 30-50) can be served even with communication based on unicast.

The WebAudio client, in contrast, scales worse than the server. The receiver needs to perform the expensive processing of audio decoding for every received packet, since they are part of different audio streams. This is expressed in equation 4.5.

$$O_{Client}(N) = N \times \{O(Receiving) + O(Decoding) + O(Mixing)\} \tag{4.5}$$

As a result of this, it is clear why the WebAudio client can only support small groups of senders of the order of 5-10. However, since conversations where many people talk at the same time is usually not convenient, one can conclude that the small groups of senders are in most cases sufficient. The limiting factors are the processing power to process all the audio packets of the various sources and the network bandwidth to receive the different streams at the same time. It should be noted that in the receiver case multicast based communication resolves neither the network bandwidth limitation nor the decoding processing problem at the receiver.

Apart from scalability problems, support for multiple concurrent audio streams at the client and server also raises other issues, namely how to distribute packets to multiple clients (without multicast communication) and how to process audio frames of several sources. Section 4.2.6 and 4.2.8 discuss how packet transmission and audio frame mixing is achieved within WebAudio.

4.1.7 Security Considerations

Special points of interest regarding the security of an application are the protocol interfaces. In the case of WebAudio, the following protocol interfaces need to be considered: the HTTP-based and RTSP-based control interface and the RTP/RTCP streaming interface.

The "open" control interface, in particular with respect to the WebAudio client, is currently a major security hole, since the client application can by remote controlled by anybody by means of HTTP or RTSP control requests. A simple technique to solve this security hole would be to restrict access to the local host. Since this approach reduces the flexibility of the application, an alternative approach is suggested: the HTTP and RTSP security mechanisms that provide secure access based on the user name and password encoded within the request URL. The WebAudio server, in contrast, intends to provide public service and therefore does not require access restriction. If limited access is required, the standard HTTP and RTSP security mechanisms are suggested as well. WebAudio client-server stream control is, to some extent, protected from manipulation by third parties. The random session identifier, generated by the server during initial stream setup and henceforth used within every RTSP-based control message, prevents hackers from interfering audio sessions.

The audio streaming interface provided by RTP/RTCP is secure due to the security mechanisms suggested by RTP and RTCP. For each audio stream, RTP uses also a randomly generated session identifier which is transmit as part of the protocol header. Receivers therefore accept RTP packets only if the session identifier matches and the sequence number is not wide of the mark. The session identifier is also used within the RTCP receiver reports. Thus, the server only accepts RTCP reports which belong to a valid RTP session.

If WebAudio is to be used for transmission of secret audio data, encryption techniques should be applied to protect against unauthorized data monitoring and insertion. In order

to secure transmission channels, IP-level security mechanisms such as IPSEC [Atk95c] are recommended.

As a result, one can conclude that the application design of WebAudio and the utilized protocols have provision for secure audio transmission.

4.1.8 Summary

This section summarizes the application design and architecture of WebAudio. WebAudio, the uni-directional real-time audio streaming application, is based on a client-server architecture. The WebAudio client and server applications provide an "open" stream control interface which enables easy Web integration. WebAudio improves stream QoS by means of resource reservation and adaptation. The multi-streaming capabilities enable the applications to be used for audio conferencing.

The operation of WebAudio can by summarized as follows: The user interface, which is invoked by the Web browser, either as a plug-in or as a helper application, controls the client application by means of HTTP or RTSP. WebAudio uses RTSP for stream control between a client and server.

The application design is modular with most modules are used within both, the client and the server. While the network interfaces of the client and server applications are very similar, the audio processing differs significantly. While the client requires an audio mixing module in order to achieve the playback of simultaneous audio streams, the server needs an audio transmission module to accomplish audio streaming to multiple receivers.

Communication between the user interface, the client and the server is achieved by numerous protocols at different levels:

- At the network level, WebAudio supports either IPv4 or IPv6. Whereas audio streaming can be achieved over both network protocols, stream control signalling is limited to IPv4.
- Resource reservation is achieved by means of RSVP. WebAudio has built-in support for standard and flow label based RSVP. Reservations are managed through the RSVP interface known as RAPI.
- At the transport level, WebAudio utilizes TCP which provides an ideal service for the transport of stream control traffic of RTSP and HTTP. UDP is used for audio streaming. Since RTP facilitates media streaming, audio data traffic is transfered over RTP-on-UDP. RTP's control protocol RTCP is used to provide a QoS feedback mechanism.
- At the application level, the WebAudio client and server can be controlled by either HTTP or RTSP. A multi-protocol control interface facilitates stream control based on both protocols.

The application interfaces of WebAudio can be described according to the supported application level protocols. HTTP based stream control allows simple control from within Web applications. Even a standard Web browser can be used as a basic user interface. RTSP based stream control enables full stream functionality within WebAudio, however, sophisticated user interfaces with support for RTSP are required. RTP which is used as the streaming protocol provides the necessary information within its protocol header to compute QoS feedback. RTCP receiver reports are used to return the QoS feedback to the server.

WebAudio applications do not incorporate a conventional user interface. Instead, they provide an "open" application control interface which offers great flexibility to external user interfaces. The HTTP application control interface has the advantage of easy Web integration of WebAudio user interfaces based on HTML, Java Script or Java. However, the sophisticated streaming control mechanism, RTSP, demands the use of programming languages such as Java, C, C++ and Pascal.

The scalability considerations lead to the following conclusions: Due to the lack of multicast support, the current version of the WebAudio server only provides support for small or moderate groups of the order of 30-50. The WebAudio client can only handle small groups of senders of the order of 5-10. However, since conversation with many concurrent speakers is not easy, small groups are in most cases sufficient.

With respect to security, it can be summarized that the application design of WebAudio and the utilized protocols have provision for secure audio transmission.

4.2 Implementation Issues

This section describes in detail the design questions and implementation problems encountered during the development phase of WebAudio. The decisions along with the reasoning and the solutions or work arounds to the problems are presented here.

4.2.1 Choice of Platform

In keeping with the application name, WebAudio aims to be used as an audio tool within the WWW. The real-time audio streaming application is therefore destined for end user systems such as PCs and user workstations. Since most of today's end user systems run Microsoft Windows – either Windows 95, 98 or NT – software releases for these platforms is important. The second most deployed platform among Internet users is currently the free Unix known as Linux. If the wide scale deployment of a software product is an important consideration, the application should be available on these operating systems.

In general, applications should presuppose as little as possible from user systems and support what the majority of user systems offer. IPv6 or RSVP, for example, cannot be

assumed since it is still rarely used outside of research labs. In contrast, if applications aim at studying experimental protocols, such as WebAudio which was aiming at exploring the impact of IPv6 on QoS based real-time streaming, experimental protocols, still unknown to the majority of users, cannot be avoided. Therefore, in the case of WebAudio, the choice of the primary implementation platform was mainly determined by the availability of the RSVP software with IPv6 support. Since the RSVP implementation with support for IPv6 was only[11] available for FreeBSD, it was the operating system of choice for the first version.

To make provision for software releases for Microsoft Windows systems, WebAudio was designed always bearing in mind that the code needs to be as platform independent as possible. Compatibility with Microsoft Windows and Linux was the main concern. Compatibility between FreeBSD and Linux is easy to accomplish, since FreeBSD and Linux are both closely related Unix implementation. Ensuring compatibility with Microsoft Windows platforms is, however, non trivial.

The process model and the interprocess communication capabilities of Microsoft Windows and Unix systems differ notably. The differences are often rooted deep down in the operating system architecture. The "fork" mechanism[12] of Unix, for example, is a mechanism that was considered but rejected since it is not supported by Microsoft Windows systems. Also multi-threaded implementations can not simply be used in a cross platform manner between Microsoft Windows and Unix systems, since the default thread support on both systems is incompatible. Although both platforms support the POSIX thread standard that offers a common API, the scheduling behavior of the different thread implementations differ greatly. This is an especially important issue with respect to performance and time critical applications, such as real-time streaming applications. This led to the decision to implement WebAudio single threaded. Concurrent processing, such as processing incoming requests and capturing data form the sound device, is achieved by preventing synchronous or blocking system calls. All read and write operations are used in an asynchronous manner. The BSD "select" mechanism[13] is used to block the process until any data or resources are becoming available for further processing.

Besides the difference in the process and thread handling, Microsoft Windows systems and Unix systems also differ significantly in the way they deal with the application *Input and Output (I/O)* (for example, the user interface and the sound device). A cross platform implementation of a graphical user interface is particularly hard since both systems support different window systems, namely Windows and X Window. This inconvenience in conjunction with the fact that WebAudio aims at easy integration within Web applications, led to the decision to abstain from a graphical user interface and instead designed an "open" control interface[14] based on HTTP and RTSP (see section 4.2.9). The interface

[11] As far as the author was aware at the time.

[12] The system call spawns a new user process with the same execution environment as the parent process.

[13] The "select" system call blocks until further data or resources become available. On return it indicates the interface which is ready for further processing.

[14] By "open" is meant that the control interface can be accessed by means of a control protocol.

to the sound device is another heterogeneity which must be resolved within WebAudio. Since the difference is also rooted deep down in the operating system, and it cannot simply be bypassed as in the case of the user interface, incompatibility has to be accepted here. To reduce the problem, an object oriented *Audio* class serving as an abstraction layer between the application and the system interface to the sound device was developed. This approach simplifies the code portability, since only the *Audio* class needs to be maintained for different platforms whereas the core implementation of the application remains the same.

As already mentioned, the first version of WebAudio was implemented on FreeBSD. The source code port to Linux was straight forward. The changes were limited to adapting a few system calls and including a few different header files. The *Unix Sound System (USS)*, provided by most Unix distributions as interface to the low-level sound device, simplified the port.

Since the author believes that support for Microsoft Windows systems is crucial for the promotion of WebAudio, support for these systems is planned in future releases. The source code port to Microsoft Windows is expected to be easily achievable due to the thorough application design and the considerations regarding platform independence. The main work is expected to be the port of the *Audio* class to support Windows sound systems, such as DirectX.

4.2.2 Choice of Programming Environment

The choice of the programming environment for the implementation of WebAudio was fairly easy. After considering all the constraints, the availability of programming languages and the advantages and disadvantages of these languages, it was obvious which programming language suits best.

The constraints which had to be considered were that (a) the programming language has to be available for FreeBSD since it was chosen as the implementation platform (see section 4.2.1) (b) the compiler must be freely available and (c) the programming environment needs to be platform independent since the WebAudio source code should be easy to port. These constraints narrow the choice of programming languages down mainly to the programming language C. Several C compilers with cross-platform support are are freely available. One of the main C compilers which is also freely available for most platforms is called GNU GCC. Apart from C, the GNU GCC compiler supports also the *Object Oriented (OO)* programming language C++.

Since C++ supersedes the functionality of C and offers in addition all the advantages of OO programming, such as class inheritance, data encapsulation, etc., it was selected as programming language for the implementation of WebAudio. OO programming also has the benefit of simplifying the reuse of source code since object classes are, if designed properly, fairly independent from the source code context.

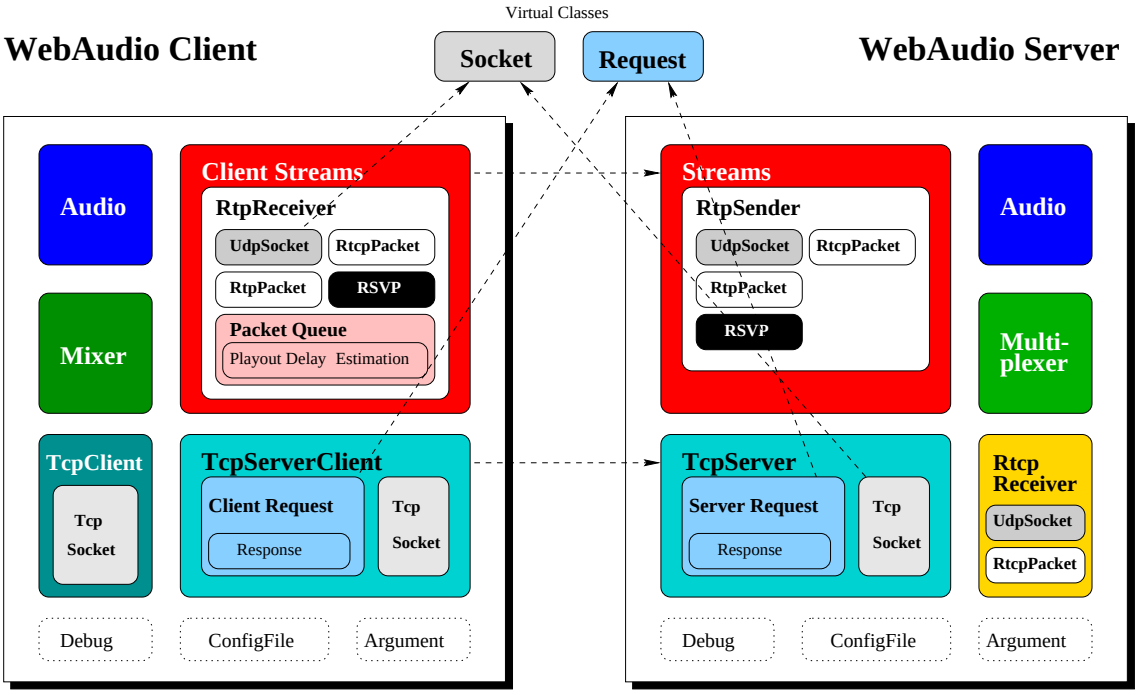


Figure 4.6: Object Class Structure of the WebAudio Client and Server Application

The object class structure of the WebAudio client and server implementations are illustrated in Figure 4.6. This object class overview illustrates how well the application structure is explained simply by viewing the object classes and their relations. The overview also shows how well the source code could be reused in the case of WebAudio due to the OO programming principles of data encapsulation and inheritance. The arrows in Figure 4.6 indicate the parents and their child classes. Most object classes are used in the client and server applications.

Summarizing, one can conclude that the choice of the object oriented programming language C++, and in particular the freely available GNU GCC compiler, has proved beneficial.

4.2.3 The WebAudio Client: Plug-In vs. Helper Application

The first implementation attempt of the WebAudio client investigated to realize the application as a Web browser plug-in[15]. Since the WebAudio client application is intended to be used within the WWW, the extension of the Web browser seemed to be an elegant way to integrate WebAudio.

[15]A plug-in is in general an application extension in form of a *Dynamic Link Library (DLL)* or shared object library that adds functionality (for example, another en-/decoder) to the application.

Implementing the client application as a Netscape browser plug-in, however, soon showed the limits of such plug-ins. Since Netscape plug-ins are executed within the Web browser's main process – no separate thread or process is executed – it is impossible to control the network communication and the sound device within WebAudio in an appropriate manner. Plug-ins simply have no sufficient control on the program flow because they are merely built of a few functions that are called upon certain Web browser events. As a result, synchronous commands such as the "select" system call cannot be used since they would block the whole Web browser process. Moreover, the main intention of Web browser plug-ins is to process data which is delivered in-band by HTTP. A typical example of a Web browser plug-in is a PDF viewer. Upon a HTTP request of a PDF document, the browser passes the `Entity Body` of the HTTP response to the plug-in which then displays the PDF document in response. In the case of WebAudio, however, the problem is completely different. The audio stream data is not transmitted in-band by HTTP. As a result the WebAudio client must be able to actively listen on a socket port. Since the plug-in has no real control on the program flow, it cannot listen on the port. One solution for this problem could be the use of a plug-in threads which actively listen on the receiving port. However, this approach failed since the FreeBSD release of Netscape's Web browser (Communicator 4.07) is not linked with thread save libraries[16] Moreover, the thread based solution is not suitable for several reasons discussed in section 4.2.1.

In addition, Web browser plug-ins have the disadvantage that they rely on the proper operation of the browser. If the browser malfunctions (for example, hangs for a moment due to an unresolved DNS lookup or an unconfirmed message box), the plug-in is also hindered from proper operation.

Another important argument against implementing the WebAudio client as a Web browser plug-in is the compatibility and support of such plug-ins. Not only are plug-ins usually developed especially for one Web browser, such as Microsoft's Internet Explorer[17] or Netscape's Communicator, they also behave differently on miscellaneous platforms (if available at all[18]), due to the different Web browser implementations. Although, for example, the plug-in interface of Netscape's Web browsers is the same on all platforms, noticeable differences in the behavior of the Communicator 4.07 for Windows 95 and FreeBSD are observable.

Helper applications, in contrast, are an alternative method to extend the functionality of Web browsers. Helper applications are stand-alone applications that are executed by the Web browser upon receipt of a certain MIME type. The `Entity Body` of the HTTP response is passed to the application through the standard input interface.

Helper applications have several advantages over plug-ins. First, they are supported by almost any Web browser – mainly due to the simple interface. Second, they are executed

[16]Such libraries are able to handle multiple threads without interfering with re-entrant system calls.

[17]In the context of Microsoft's Internet Explorer plug-ins are usually called ActiveX controls.

[18]Microsoft's Internet Explorer is not available for many other operating systems apart from the Microsoft Window systems.

in a separate process. Thus, the application has full control over the program flow which greatly simplifies the problem of concurrent processing of time-critical operations, such as listening on a receiver port while decoding and playing back the audio data in time. Third, since helper applications are executed independently from the Web browser process, problems caused by differences in cross platform browser implementations do not arise.

The main drawback of using helper applications is that the Web integration is not as thorough. While plug-ins are up-called on relevant Web browser events (for example, window closed, page unload, etc.), helper applications are not informed. This disadvantage, however, is less significant in the context of WebAudio since the client application offers a HTTP based control interface. Java Script, for example, can be used to forward Web browser events to the client application by means of HTTP-URL requests.

In retrospect, the discussion of this section has shown that the implementation of the WebAudio client as a helper applications has significant benefits over the implementation as a Web browser plug-in.

4.2.4 Real-Time Audio Processing and Task Scheduling

During the implementation phase of WebAudio, problems regarding the scheduling of concurrent tasks (for example, reading from the socket and writing data to the sound device) with respect to their time constraints were experienced.

In order to illustrate the problem, a short description of the task scheduling and the strict time constraints of real-time audio is provided. Interactive real-time audio streams use, according to the discussion in section 3.1.1, very small packets, or in other words only few audio samples, to minimize the end-to-end delay caused by packetization. In practice, real-time audio tools often use a packet size that encompasses as little as 20 ms or 40 ms audio. From now on this time is referred to as *Packet Playout Time (PPT)*. The PPT is usually a multiple of the audio frame length (see also section 3.1.1). As a result, the WebAudio server, on the one hand, needs to capture, encode and send an audio packet within any PPT period. The client, on the other side, needs to receive, decode and playout the audio packet within this interval. The problem becomes obvious when the process scheduling granularity of common operating systems is examined. Unix systems usually have maximum scheduling units of approximately 5-20 ms. This means that all other "runnable" processes[19] are scheduled (at the most the maximum scheduling unit) before the WebAudio process is activated again. Fortunately most processes are in blocking state and "runnable" processes are often consuming only small amounts of their processing time such that the scheduling delay of the WebAudio process is usually short enough to meet the strict time constraints of real-time audio. However, if the machine load is high or time

[19] All processes, which are neither "blocked" while waiting for a resource nor "stopped", are referred to as "runnable".

consuming operations (for example, disk access) are scheduled, deadlines of the real-time processing within WebAudio might be exceeded.

An important implementation issue is that the task that needs to be processed next is always scheduled first. Rather than spending the processing time on a task which is ahead of its time schedule, the most urgent task should be processed first. For example, rather than reading the whole receiving buffer or en-/decoding several packets of a queue at once, it is more important to play back the packets whose playout time have exceeded.

Since the processing of the audio data within the WebAudio client and server differ significantly from each other, they are considered separately here.

The implementation of the server application regarding to scheduling is less critical than the client application. Since the data flow is predestined by the audio capturing module – only one time-critical module – the application can simply block on the sound device until a new audio frame can be read. As soon as the "read" call returns, the frame is encoded. If sufficient audio frames for a packet are captured, the packet is sent to all receivers of the stream. Before the server process blocks again (while waiting for the next audio frame to be read), it processes pending stream control requests.

The client application, in contrast, requires a more careful design with respect to task scheduling. The data flow is predestined by the audio playback module and the RTP receiver module – two independent, time-critical modules which have to be processed concurrently. The audio playback module has to playout a new frame periodically. The time interval is determined by the frame length. Since the lack of audio samples in the sound device playout buffer immediately leads to disturbing crackles in the signal, the client has to make sure that the playout buffer never runs empty. The second time-critical module is the RTP receiver. As soon as a new audio packet arrives at the UDP port, the packet must be unpacked, decoded and queued. The playout time, depending on the current playout delay estimation, is assigned. As a result, the client's task scheduler must be designed such that it guarantees to meet the time constraints of both time-critical modules. The use of blocking system calls, while waiting for new data being received, on the one hand, and waiting for the sound device to play the next frames, on the other hand, would not provide concurrency.

The asynchronous "select" mechanism cannot be used within the audio playback module since the system call would return as soon as a few samples could be written to the sound device. This, however, would be virtually at any time, since the sound device permanently plays audio samples from the buffer. Recent work in this area [Riz97] extended the sound device driver of USS such that the synchronous "select" behavior can be programmed to return only if a minimum threshold is under-run. However, having a competing "select" call for both time-critical modules, is not a good solution. The module invoking the blocking "select" call first would block the process until the resources becomes available.

Figure 4.7 illustrates how the time-critical task scheduling problem is solved within the WebAudio client.

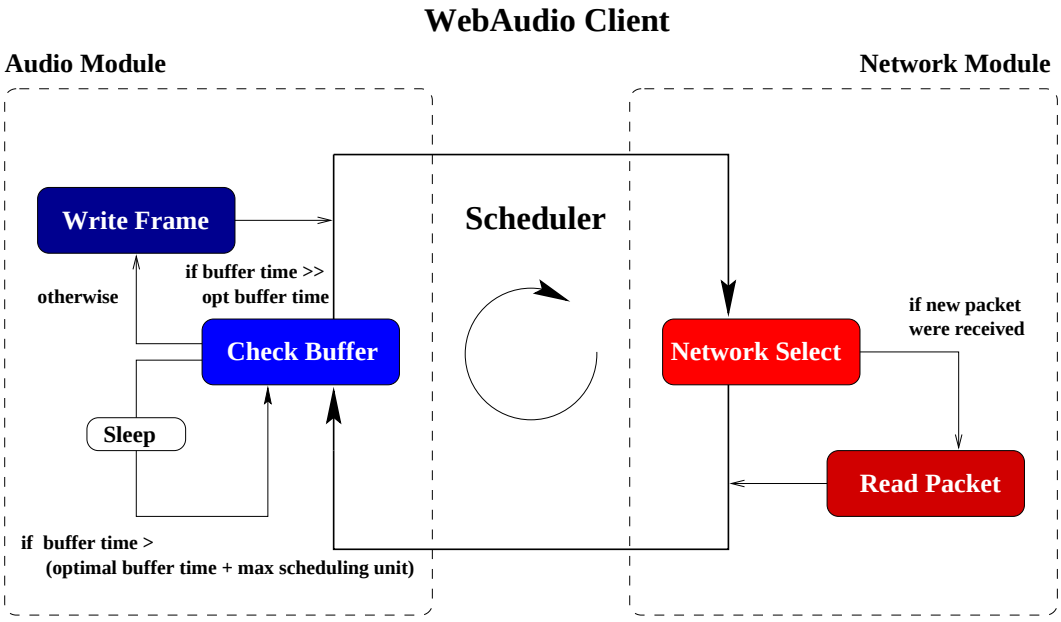


Figure 4.7: Time-Critical Task Scheduling within the WebAudio Client

The client implementation makes use of the sound device driver feature that determines the amount of audio samples in the playout buffer [20]. Based on the scheduling behavior of the system, the client computes dynamically the optimal buffering time of frames in the sound device such that the buffer hardly runs empty. This sophisticated algorithm "guarantees" that the audio signal is not permanently disturbed due to buffer under-runs caused by scheduling irregularities in the client system.

The calculation of the optimal buffering time is accomplished by the following algorithm: based on the past scheduling behavior and a threshold percentage $T_{success}$, the optimal buffering time is estimated such that $T_{success}$ percent (usually $T_{success} > 95\%$) of the past scheduling cycles were re-scheduled in less or equal time than the optimal buffering time. This adaptive mechanism estimates the optimal buffering time depending on changes in the scheduling behavior. Such changes are, for example, caused by an increase or decrease of the processing load. The adaptive behavior guarantees optimal performance since it tries to keep the buffering delay and thus the total end-to-end delay as small as possible. A more detailed description of this adaptive buffering time estimation is provided in section 4.2.5 where the same algorithm is used for the estimation of the optimal packet playout delay.

Experiments with FreeBSD and Linux on different system architectures, such as Intel Pentium II, Mobile Pentium 166 and Intel Pentium 90, have shown that the adaptive buffering time estimation operates well for these systems. Debug traces indicated that the

[20] The Unix device control call "ioctl" allows to request the number of bytes that can be written and the total buffer size of the low-level sound device driver.

algorithm properly adapts to different levels of the system load. Since process scheduling behavior is highly dependent on the operating system, the algorithm might have to be adjusted when the application is ported to Microsoft Windows systems. The buffering time estimation algorithm resides in the *Audio* class which has to be specially ported in any case.

In retrospect, one can summarize that the implementation of the task scheduling within WebAudio, and in particular the client application, was not straightforward but has been successfully solved. Using an adaptive buffering time estimation to compensate for the dynamics within process scheduling has been proven to be a useful enhancement.

4.2.5 Receiver Buffering

Receiver buffering is required within real-time streaming applications in order to compensate for the jitter caused by scheduling irregularities in the sending system and the delay variations experienced by individual packets along the network transmission path (see section 1.2.6 and 3.1.4 for further discussions).

Since the scheduling and jitter varies highly over time due to changes in the processing and the network load, it is advantageous to use a dynamic receiver buffering mechanism. The enhanced adaptive playout delay estimation algorithm used within WebAudio is based on the work which has been done at ... . The algorithm dynamically adapts the buffering time of the packets, or in other words their playout delay, by considering the jitter experienced by former packets. The receiver buffering module maintains two playout delay values: first, the optimal playout delay which is constantly changed upon arrival of new packets, and second, the currently active playout delay. The latter value is used as long as the receiver buffer does not under-run. When the receiver buffer runs empty, the active playout delay value is adjusted to the optimal playout delay estimate. The receiver buffer runs empty when no packets arrive within the buffering time due to either packet loss, high end-to-end delays, or silence periods. Changing the active playout delay while packets are still in the buffer would result in interruptions of the output signal due to "inserting gaps" or "overlapping audio frames" which leads to buffer overflows in the sound device. However, when the receiving buffer runs empty, a change to the active playout delay is not noticeable, and therefore, this approach is applied.

The playout time of incoming packets is determined by the timestamp of the RTP header and the currently active playout delay according to equation 4.6. When the $PlayoutTime$ of a packet exceeds, the packet is dequeued and decoded (see section 4.2.8).

$$PlayoutTime = PacketTimestamp + ActivePlayoutDelay \tag{4.6}$$

Before the extended playout delay estimation which has been developed in the context of this work is described, the original playout delay estimation from University of Massachusetts (UMASS) [MKT98] is briefly explained. The algorithm records a large number

of past packet end-to-end delays. It is suggested that the last 10000 packet delays are taken into consideration. The algorithm calculates the optimal playout delay based on a threshold percentage called $T_{success}$. A value of $T_{success} = 95\%$ would result in a playout delay such that at least 95% of the packets arrive before their playout point is exceeded whereas 5% of the packets arrive late.

Since packets sometimes experience huge delays (and therefore huge delay variations) due to temporary congestion, the algorithm has a delay spike detection mechanism that allows spike delays to be specially considered. If a delay spike is detected, the algorithm simply suggests to use the delay of the first packet. This simple technique results in good results during "spike" periods, since the analysis of Internet traffic has shown that the delays of subsequent packets of a "spike" decrease linearly [MKT98]. Delays of "spike" packets are not recorded and considered in the standard delay estimation to prevent these delays from making the optimal delay estimation ambiguous. As soon as the end of a spike is detected, the normal playout delay estimation is continued.

Although [MKT98] shows that the UMASS algorithm performs well, two problems were recognized. Both problems are solved successfully in the improved playout delay estimation algorithm used within WebAudio. The first problem of the original algorithm is that a large amount of memory is required simply to store the past delay values. The authors suggest to consider the last 10000 delays for the playout estimation. These delays encompass a time period of about 3.5 minutes if a PPT of 20 ms is assumed. To have more stable results, it is necessary to consider the packet delays of the last few minutes. This prevents the results from being fuzzy due to temporary irregularities in the network delay. The algorithm developed within WebAudio records only the last 1000 packet delays. In order to account for such temporary irregularities, an array (a size of ten is suggested) is used to store the last optimal playout delays. The optimal playout delay is then calculated by weighting recent optimal playout delays based on their temporal ordering. (see equation 4.7).

$$PlayoutDelay = \frac{\sum_{i=1}^{N} (i \times PastPlayoutDelay[i])}{\sum_{i=1}^{N} i} \tag{4.7}$$

This formula is also known as the decay average calculation. It should be noted that the formula assumes that the array of past optimal playout delays is sorted according to the time when the delay estimates where computed. The oldest delay estimate is stored in the first field.

The advantages of this approach are that less memory is required to take into account a long history of past packet delays and that the playout delays experienced in the recent past are weighted higher. This weighted approach to compute the optimal playout delay also solves the second problem of the original algorithm. It did not respond well to trend changes in the network's QoS. The original algorithm needed several minutes to react to

a QoS change in the network. The weighted approach in our solution, however, improves the responsiveness by weighting the latter results higher than the former.

Experiments with WebAudio have shown that our improved playout delay estimation algorithm performs well and requires only a fraction of the memory of the original algorithm. Moreover, it has been shown that the algorithm adapts faster to trend changes in the network without loosing accuracy during temporary irregularities in the network delay.

4.2.6 Audio Capturing, Encoding and Packet Transmission

The audio capturing and encoding is processed within the *Audio* class. This class is especially developed for each platform to account for differences in the sound device interface and the audio codec support of the operating systems. The class provides a uniform interface for the WebAudio applications. Thus, it can be seen as an abstraction layer for the low-level sound system provided by the operating system.

According to the discussion in section 4.2.4, the WebAudio server reads the sound samples synchronously from the sound device. If a complete audio frame is read successfully, the transmission module is called by passing a pointer to the new audio frame. The transmission module encodes the audio frame into the set of audio formats currently in use. As soon as enough frames for a packet of this audio format are encoded, the RTP header is attached and the packet is sent to every registered WebAudio client.

Figure 4.8 illustrates how the audio capturing, encoding and packet transmission is processed by the WebAudio server.

The transmission mechanism used to support group communication or even audio broadcast obviously does not scale in the number of group members. Section 4.1.6 provides an in-depth discussion on the scalability issues. The maximum number of streams processed by the WebAudio server is limited by the processing power of the server machine and the available bandwidth on the network. Within the period of one PPT, the server must be able to encode and transmit all the packets. Since the encoding of the audio frame needs to be done only once for each audio encoding used, only the simple processing of sending multiple packets does increase linearly with the number of concurrent streams (compare with equation 4.4). As a result, normal user workstations can handle a moderate number of streams. Experiments with WebAudio have shown that our test machine, a Pentium II with 233 MHz running FreeBSD, can easily handle 40-50 stream simultaneously.

4.2.7 Audio Codecs

In the first version of WebAudio only two audio encodings, namely *PCM (Pulse Code Modulation)* and *GSM (Global System for Mobile communication)* [C+89] are supported.

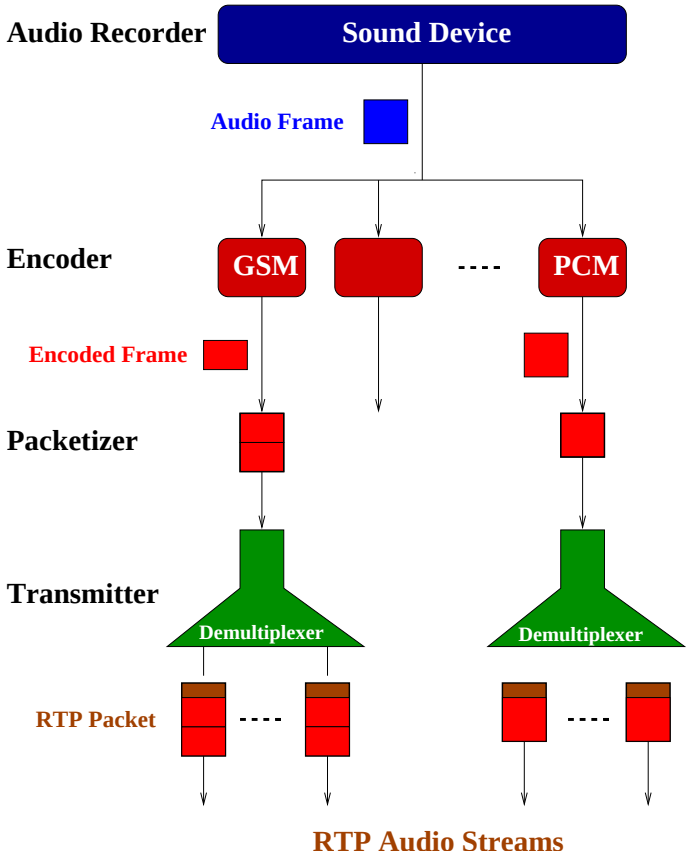


Figure 4.8: Audio Capturing, Encoding and Packet Transmission within the WebAudio Server

In future releases of WebAudio several new codecs are planned to be included to cover the full range from low-bandwidth voice to high-quality sound codecs.

Adding new audio codecs to WebAudio is fairly simple, since the *Audio* class is designed bearing in mind that new codecs will be added. The code changes are mainly restricted to the *Audio* class where the encoding and decoding function calls of the individual codecs reside. In addition, the availability of a new codec must also be introduced to the overall application by announcing a new audio resource.

Since the support of audio codecs is highly dependent on the operating system [21], the *Audio* class has provision to easily add new audio codecs. The audio encoding and decoding functions reside in the *Audio* class since this class needs to be ported to each individual platform in any case.

[21] Some systems, for example Microsoft Windows, have a wide range of audio codecs included in the operating system distribution.

4.2.8 Frame Decoding, Mixing and Audio Playback

The WebAudio client consists, apart from the receiving module, of an audio mixing and playback module. Since the WebAudio client is intended for use as a communication tool for small groups, it must be capable of receiving and processing multiple audio streams simultaneously.

Standard sound devices, however, support only simplex or duplex sound I/O. Simplex mode is limited to one channel which is shared time-wise for sound input and output whereas duplex mode allows playing and recording of sound simultaneously. Applications which need to playout multiple audio streams at the same time must implement a sound mixer. A sound mixer simply merges audio samples of multiple streams into one sample representing the overlayed sound. Formula 4.8 determines how the mixed audio sample a' is computed from the different audio samples a_n of the N streams:

$$a' = \frac{\sum_{n=1}^{N} a_n}{N} \tag{4.8}$$

The processing of the audio stream at the WebAudio client can be described as follows. First, when the receiving module gets a new packet, the module assigns the playout time based on the currently active playout delay of the stream (see section 4.2.5) and then puts the packet in the receiver buffer. The playback module checks periodically the buffer of the sound device in order to prevent it from running empty. This time-critical process is described in detail in section 4.2.4. If the sound device needs another frame of samples, the playback module checks the receiver buffers for packets whose playback point is exceeded. If multiple audio streams are received simultaneously, all packets whose playout time is exceeded need to be mixed before the playback. Therefore, the mixer initiates first the decoding of the audio packets by calling the individual decoding methods within the *Audio* class. After decoding the frames of the different streams, they are mixed together and then passed to the sound device. If no packet is ready to be played back, the last audio frame is played again, up to three times, and after that, silence frames are inserted to fill the gap in the audio signal. In future versions of WebAudio, support for playing back background noise rather than silence is planned since this is commonly perceived to be more natural [H$^+$95].

Figure 4.9 illustrates how the packets of the receiving buffer are decoded, then mixed and finally passed to the sound device.

Although the audio mixer enables the WebAudio client to receive multiple streams simultaneously, and hence, allows multi-user conferencing, it does not scale. The maximum number of streams is mainly limited by the processing power of the end system. The client host must be able to decode and mix all received packets of the different flows within one PPT. If the number of concurrent flows exceeds the maximum limit, the client cannot complete the processing within one PPT period. However, according to the discussion in

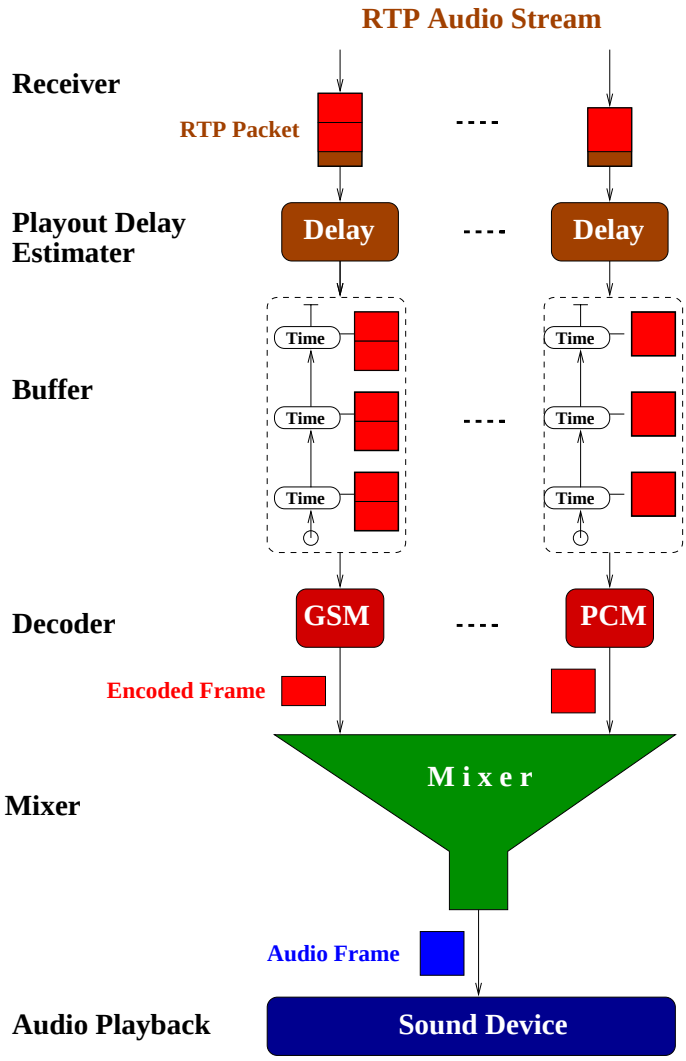


Figure 4.9: Packet Receiving, Audio Decoding, Frame Mixing and Sound Playback within the WebAudio Client

section 4.1.6, two-way communication among several users is only useful for small groups. Experiences with WebAudio have shown that a client running FreeBSD on a Pentium II with 233 MHz can easily handle 5-10 streams at the same time.

4.2.9 Multi-Protocol Control Interface

With respect to the discussion in section 4.1.4 WebAudio provides two "open" control interfaces, namely HTTP and RTSP. Support for the two different stream control protocols might not seem worth the extra development work. However, HTTP and RTSP are so alike that the additional source code to support both protocols merely increases the size of the

control interface module. Since RTSP was designed in the style of HTTP, the protocol syntax is so similar that a standard HTTP parser can be used to parse RTSP. Merely a new protocol type and a few additional header fields need to be added. Table 4.3 shows the common request format of both protocols (compare also with section 2.4.1 and section 2.4.2).

Request	=	Request-Line *(General-Header | Request-Header | Entity-Header) CRLF [Entity-Body]
Request-Line	=	Method SP URL SP Protocol-Version CRLF

Table 4.3: Syntax of RTSP and HTTP Requests

The `Protocol-Version` in the `Request-Line` is used to identify the protocol: HTTP or RTSP. The `Method` contains either the HTTP method GET[22] or the RTSP methods (see Table 2.9). The `URL`, used to identify the audio resource, varies slightly depending on the protocol.

The syntax of the response messages of HTTP and RTSP is also very similar. The differences are that the response codes and messages vary – RTSP has a few additional response codes to indicate request processing problems (compare with Table 4.1 and 4.2) – and a few additional header fields (for example, the `Session` and `Transport` field) are introduced.

Moreover, apart from the protocol syntax of HTTP and RTSP, the protocol semantics of these protocols are also very similar. It is described here, how WebAudio manages the different semantics of both stream control protocols. Since the stream control functionality of RTSP supersedes the functionality of HTTP with respect to stream control (compare with section 2.4.3), WebAudio simplifies the control message processing by mapping HTTP control requests onto RTSP requests. Thus, when the WebAudio server or client receives a HTTP based control request, it maps the request onto the corresponding RTSP request(s). Table 4.4 shows the HTTP→RTSP mapping of the HTTP stream control requests that are currently supported by WebAudio.

HTTP Request	→	**RTSP Request(s)**
PLAY	→	SETUP PLAY
STOP	→	TEARDOWN

Table 4.4: Mapping of HTTP Requests to RTSP Requests

[22]WebAudio only uses the GET method of HTTP.

The following example shows how the WebAudio client makes use of the HTTP→RTSP mapping in the processing of HTTP requests received from the user interface. The HTTP requests are transformed into RTSP requests and forwarded to the WebAudio server.

The HTTP request

```
GET /liveaudio/gsm?PLAY&SERVER=webaudio.lancs.ac.uk:4444&
 PORT=8001 HTTP/1.0
[CRLF]
```

is mapped onto the following RTSP requests

```
SETUP rtsp://webaudio.lancs.ac.uk:4444/liveaudio/gsm RTSP/1.0
CSeq:  120
Transport:  RTP, unicast, client_port=8001
```

and

```
PLAY rtsp://webaudio.lancs.ac.uk:4444/liveaudio/gsm RTSP/1.0
CSeq:  121
Session:  3213221
Range:  npt=0-
```

The `SERVER` parameter of the HTTP request indicates the RTSP server which offers the audio resource `/liveaudio/gsm`. It is therefore used to specify the host name in the RTSP-URL. The `PORT` parameter indicates the receiving port of the client application. As a result, it is used within the `Transport` field of the RTSP request to negotiate the transport mechanism and the protocol ports. If the `SETUP` request could be processed successfully, the `PLAY` request, specifying the time range `Range` for live sources `npt=0-`[23], is initiated.

The HTTP→RTSP request mapping has the following implications for the response processing: First, in the case of an unsuccessfully processed request, the WebAudio client needs to map the RTSP error code to the "closest" HTTP error code in meaning and return it to the user interface. Second, if a single HTTP request is mapped onto two RTSP requests, the WebAudio client has to return a HTTP error if either of the RTSP requests failed; otherwise the HTTP acknowledge (response code 200) is sent back.

Summarizing, one can conclude that the multi-protocol control interface within WebAudio was very easy to accomplish due to the similarity of HTTP and RTSP regarding syntax and semantics. The HTTP→RTSP request mapping facilitated the implementation significantly.

[23]This specifies the time range from now onwards.

4.2.10 Summary

This section summarizes the implementation issues that were encountered during the development phase of WebAudio.

Choice of Platform: Even though the first version of WebAudio only supports FreeBSD and Linux, the client and server applications are designed portability to Microsoft Windows operating systems in mind. For platform independence, WebAudio (1) is based on a single threaded processing model, (2) provides an "open" control interface that simplifies cross-platform user interfaces, and (3) uses its *Audio* object class as an abstraction layer to the low-level sound device and OS dependent audio encoders.

Choice of Programming Environment: WebAudio is implemented based on the programming language C++. The freely available GNU GCC compiler was used. The OO principles of inheritance and data encapsulation have proven to be particularly beneficial with regard to re-usability of source code.

Plug-In vs. Helper Application: Implementation of the WebAudio client as a helper application has several advantages over a Web browser plug-in: (1) helper applications have full control on the program flow, (2) they do not rely on the proper operation of the Web browser, (3) helper applications are supported by most Web browsers, and (4) differences in cross platform browser implementations are not an issue.

Real-Time Audio Processing and Task Scheduling: Task scheduling within WebAudio, and in particular the client application, was not straight forward, but has been resolved successfully. Adaptive buffering time estimation to compensate for the dynamics of process scheduling has proven to be a useful enhancement.

Receiver Buffering: As a receiver buffering mechanism, an extended version of the adaptive playout delay estimation of [MKT98] was developed which requires only a fraction of the memory of the original algorithm and adapts faster to trend changes in the network without losing accuracy.

Audio Capturing, Encoding and Packet Transmission: Transmission mechanisms based on single unicast streams for every receiver do not scale. However, since the transmission module encodes the audio frames only once for every audio format in use, merely the simple processing of sending multiple packets is required. As a result, normal user workstations can handle a moderate number of streams.

Audio Codecs: Even though WebAudio currently supports only PCM and GSM, the Audio class has provision for new codecs. In future releases, support for the full range from low-bandwidth voice to high-quality sound codecs is planned.

Frame Decoding, Mixing and Audio Playback: Although the audio mixer enables the WebAudio client to receive multiple streams simultaneously, it does not scale. Since the client must decode all received packets individually, the maximum number of streams is primarily limited by the processing power of the end system. Thus, normal user workstations can only handle a small number of streams.

Multi-Protocol Control Interface: The multi-protocol control interface, offering great flexibility to user interface developers, was very easy to build due to the similarity of HTTP and RTSP regarding syntax and semantics. The HTTP→RTSP request mapping facilitated the implementation.

4.3 Summary

This chapter describes the design and architecture of the WebAudio server and client and provides a discussion on the implementation issues that were encountered during the development phase.

The main characteristics of the real-time audio streaming application WebAudio can be summarized as follows:

- WebAudio is based on an asymmetric, client-server based architecture which enables simplex audio streaming.
- The application was carefully designed emphasizing source code portability among Unix and Microsoft Windows operating systems.
- Support for multiple audio encodings is provided.
- WebAudio operates within IPv4 and IPv6 networks.
- Network level QoS according to the IntServ architecture is supported based on the resource reservation protocol RSVP.
- Audio streaming is accomplished based on RTP. QoS feedback is provided by means of RTCP.
- Adaptive receiver buffering which takes current network QoS characteristics into account, is applied to compensate for the jitter.
- The WebAudio client and server are capable of managing multiple streams simultaneously. The server provides service from small to moderate groups, whereas the client is limited to small groups.
- The “open” multi-protocol control interface enables out-of-band stream control based on HTTP and RTSP.

- WebAudio user interfaces which control the applications through the control interface can be easily integrated within Web applications.

The important findings of the development and implementation of WebAudio can be summarized as follows:

- The implementation based on the object oriented language C++ was in particular profitable as regards the re-usability of source code within the client and server. Most object classes could be used in both applications.
- The multi-protocol control interface was very easy to accomplish due to the similarity of HTTP and RTSP with respect to syntax and semantics.
- The enhanced adaptive playout delay estimation which was developed within WebAudio requires only a fraction of the memory of the original algorithm and adapts faster to trend changes in the network without losing its accuracy.
- The choice of implementing the client as a helper application has proven to be beneficial especially with respect to the processing of time-critical audio data.

Chapter 5

Experiments

This chapter presents a series of experiments performed with the WebAudio application. These experiments can be divided into three groups. The first group of experiments verifies the proper operation of WebAudio within various network environments. The QoS feedback mechanism offered by the RTP/RTCP implementation in WebAudio is exploited to analyze the network QoS received by the application when used in these network environments and along different transmission paths. The second group of experiments validates the resource reservation support of WebAudio. The extended RSVP implementation which deploys the IPv6 flow label is used within the experiments. The third group of experiments includes a theoretical analysis and a set of measurements showing the difference of packet classification[1] based on RSVP for IPv4, IPv6 and IPv6 with flow label support. A performance comparison of all three packet classification approaches shows the benefits of IPv6 flow labels.

5.1 QoS Analysis in various Network Environments

This section examines the network QoS received by WebAudio when used along different transmission paths and operated in different networks, namely in the public Internet (IPv4), in pure IPv6 networks and in the experimental, virtual IPv6 network (6Bone).

The experiments in real IPv6 networks were limited to experiments within the IPv6 testbed at Lancaster University [2], since today's public networks do not offer IPv6 network communication across wide-area networks. The 6Bone, however, is especially designed for IPv6 experiments across wide-area networks. It is currently used to test new software releases that operate on IPv6. Since the 6Bone is only a virtual network that inter-connects IPv6

[1] Packet classification is according to the discussion in section 2.3.1 one of the required tasks within resource reservation.

[2] IPv6 Resource Centre at http://www.cs-ipv6.lancs.ac.uk/

networks by tunneling the IPv6 traffic across the IPv4 Internet, it is not suitable for network performance measurements. Therefore, the 6Bone experiments presented here do not aim to compare the network performance with the other approaches. It is obvious that the 6Bone performs equally or even worse in comparison to IPv4 on the same path since IPv6 packets are simply tunneled through IPv4. The 6Bone experiments are included to validate the proper operation of WebAudio across wide-area IPv6 networks. Furthermore, a comparison of network QoS measurements along the same network path but via different protocols, namely IPv4 and IPv6, is interesting (see section 5.1.2). The 6Bone experiments are limited to one remote site since access to a Linux or FreeBSD host at a remote 6Bone site was only available at BT Labs, Ipswich (U.K.). The experiments on wide-area IPv4 networks were performed between Lancaster University and the remote sites at the University of Ulm (Germany) and BT Labs.

The experiments presented here demonstrate that WebAudio operates well in IPv4 and IPv6 network environments. The resource reservation capabilities of WebAudio are demonstrated later in section 5.2.

General Experimental Setup

The setup for the different experiments presented within the next sections is fairly similar. Figure 5.1 illustrates the general setup. The WebAudio server, on the one hand, operates on a local machine, a standard PC running FreeBSD 2.2.5, of the IPv6 testbed at Lancaster University. The machine is configured to support both IPv4 and IPv6 as network protocols. This enables the operation of WebAudio in both modes. The WebAudio client, on the other hand, is executed on different Linux PCs, depending on the experiment, locally at Lancaster or remotely in Ulm or Ipswich.

Netscape's Communicator is used to provide a user interface for stream setup and control during the experiments. The Web browser is executed on the local test machine. The WebAudio client is controlled through the HTTP interface (see section 4.1.4.1). The client in contrast uses the RTSP control interface to initiate and teardown the streams.

Measurements and Evaluation

The WebAudio client and server are configured such that they log the transmission traces during the experiments to their local file system. Whereas the WebAudio client records the inter-arrival time of every received RTP packet, the server records the QoS feedback information provided by the periodic RTCP receiver reports (see section 4.1.4.3). The feedback encompasses the transmission delay of the last packet received, the smoothed jitter[3], the total number of lost packets, and the fraction of packet loss within the report period.

[3]The smoothed jitter is calculated according to equation 4.2.

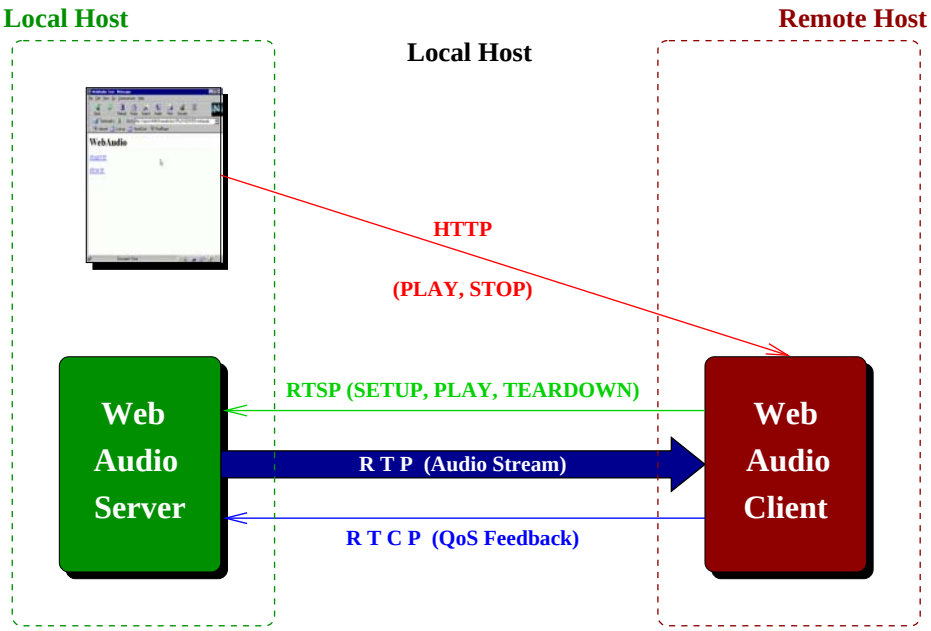


Figure 5.1: The General Experimental Setup

The duration of each experiment is set to be approximately 20 minutes which is equivalent to sending about 60000 audio packets when the PPT (Packet Playout Time) is set to 20 ms. This provides a sound basis for the network QoS comparison between the various experiments. Since transient network irregularities, such as temporary congestion, usually remain for only a couple minutes [MKT98], the suggested experiment duration should prevent the measurements from being fuzzy.

Based on the QoS measurements recorded by the WebAudio client and server for the different experiments, a number of graphs presenting the network QoS with respect to the QoS parameters: jitter, packet loss and packet inter-arrival time, are generated. These graphs, presented within the next sections, provide the basis for the comparison and evaluation.

5.1.1 IPv4 Networks

Since the experiments with remote sites in Ipswich and Ulm take place in the public Internet, the measurements depend highly on the time when they are carried out. It is known that the network QoS characteristics of the Internet vary strongly depending on the day and the time of day. As a result, each experiment was performed twice: the first time on a Sunday at around 10 pm, this will be referred to as *off-peak* time; the second time on a Monday at around 2 pm, this is called *peak* time. Extreme differences in the network QoS are expected between the *peak* and *off peak* time experiments since the network load varies highly at these times.

Figures 5.2 and 5.3 present the inter-arrival times of the packets transmitted along the different routes from Lancaster to Ipswich and Ulm. During *peak* time the inter-arrival times vary significantly more than during *off peak* time. The graphs also illustrate the differences in the destination locations. Whereas the route to Ulm has approximately 17 hops and average round-trip delays of the order of 50 ms, the route to Ipswich has only 11 hops and round-trip delays of the order of 20 ms. According to the discussion in section 1.2.6, more intermediate network hops on a transmission path increase the likelihood of packet clustering which in turn has a negative impact on delay variation. The observations of these experiments affirm this hypothesis. The variation of the inter-arrival time of the packets to Ulm are significantly higher than those to Ipswich.

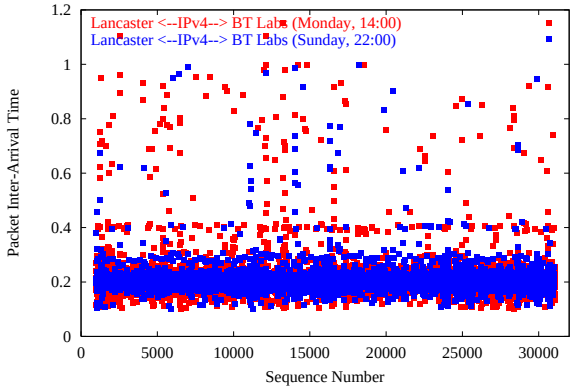


Figure 5.2: Packet Inter-Arrival Times from Lancaster to Ipswich

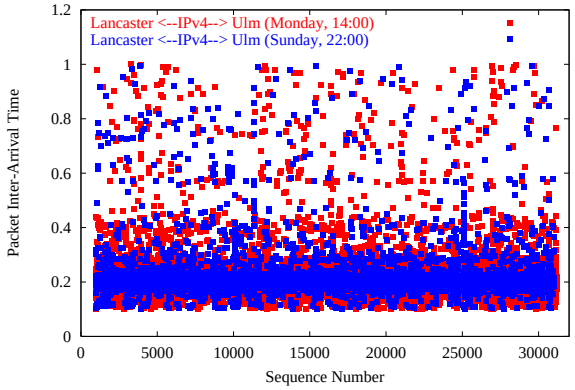


Figure 5.3: Packet Inter-Arrival Times from Lancaster to Ulm

Figures 5.4 and 5.5 illustrate the smoothed jitter experienced by the audio packets along the transmission paths to Ipswich (red) and Ulm (green). These graphs again confirm the observations above.

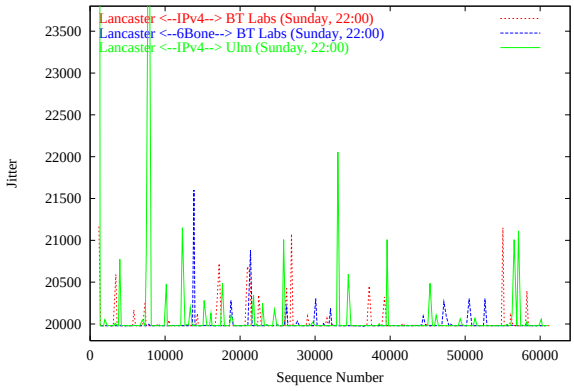


Figure 5.4: *Off Peak* Times Delay Jitter

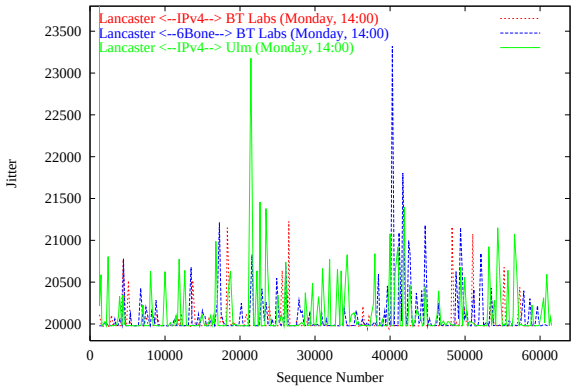


Figure 5.5: *Peak* Times Delay Jitter

The graphs also show the impact of the network load on the jitter. The average jitter experienced by the packets is substantially higher during *peak* time. High network load

causes temporary congestion and leads to high dynamics in router queues what results in high delay variations.

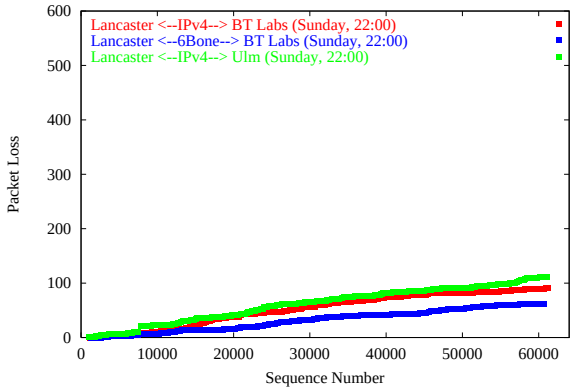


Figure 5.6: *Off Peak* Packet Loss

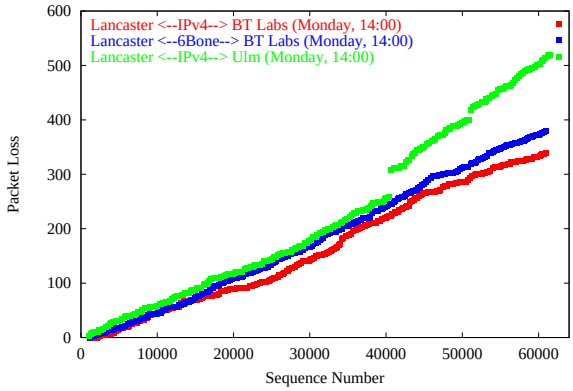


Figure 5.7: *Peak* Packet Loss

Figure 5.6 and 5.7 show the progressions of the total packet loss experienced by the audio streams on the different paths. Again, the difference in the number of hops on the transmission paths is reflected in the graphs. More intermediate network hops in the delivery path increase the likelihood of packet loss. The stream to Ipswich (red) experienced, as a result, less packet loss than the stream to Ulm (green). However, the times of the experiments make a significant difference. If the network is highly loaded, as in *peak* times, the probability of loss is much higher due to frequent congestion in the network. The packet loss trace of the experiment with the remote site at Ulm even shows remarkable loss bursts. Under these circumstances, the number of network hops has a strong impact on the packet loss rate since the likelihood for congestion in any intermediate node increases with every hop.

5.1.2 6Bone Networks

The network configuration for the 6Bone experiment between Lancaster University and BT Labs (Ipswich) is presented in Figure 5.8. It illustrates how the virtual IPv6 link between Lancaster and Ipswich is achieved. The 6Bone routers `cisco` and `btlabs-r` tunnel the IPv6 packets through the IPv4 network between these sites. As a result, an IPv6 route trace from **spock** to **vodclient** shows only 3 hops even though the physical network path from Lancaster to Ipswich is the same as in the case of IPv4.

```
traceroute6 to 3ffe:2c00::60:9777:71ea (3ffe:2c00::60:9777:71ea) from
3ffe:2101:0:800:260:8ff:fe66:39e1 (3ffe:2101:0:800:260:8ff:fe66:39e1),
30 hops max, outgoing MTU = 1480
 1 cisco (3ffe:2101:0:800::1) 7 ms * 4 ms
 2 btlabs-r (3ffe:2101:0:ffff::2) 42 ms * 42 ms
 3 3ffe:2c00::60:9777:71ea (3ffe:2c00::60:9777:71ea) 44 ms 41 ms 40 ms
```

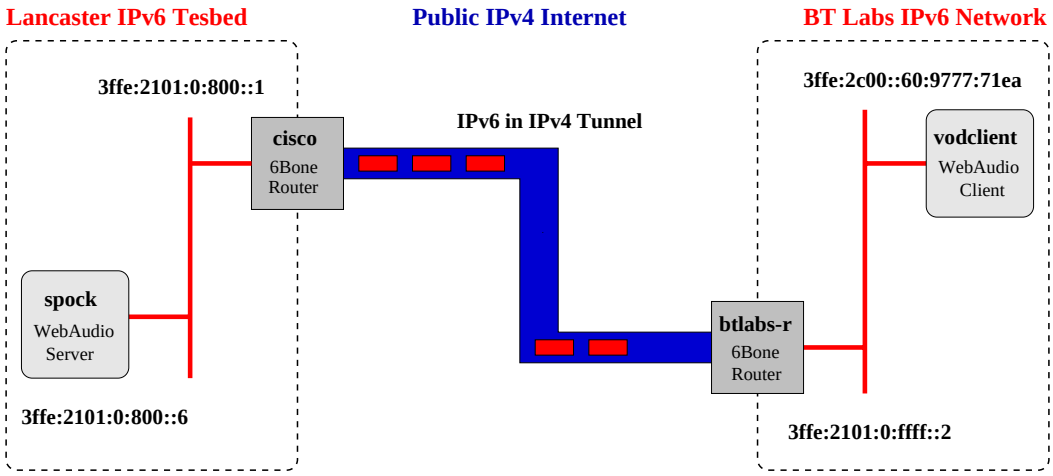


Figure 5.8: The 6Bone Link between the Lancaster IPv6 Testbed and BT Labs

Although most results of the 6Bone experiments were already presented in comparison with the experiments on IPv4 (see Figures 5.6, 5.7, 5.4 and 5.5), they are discussed here.

The network QoS received by the audio packets in the virtual 6Bone connection from Lancaster to Ipswich is closely related to the QoS experienced by the packets on the IPv4 connection. Even though the smoothed jitter and the packet loss are slightly higher in the 6Bone experiment, the traces have similar progressions. The explanation for this incident is simple. The fact that the audio streams are both delivered along the same physical network links account for the similarity of the network QoS measurements. Since the IPv4 tunneling of IPv6 packets requires extra processing within the 6Bone routers, the experienced network QoS is slightly worse.

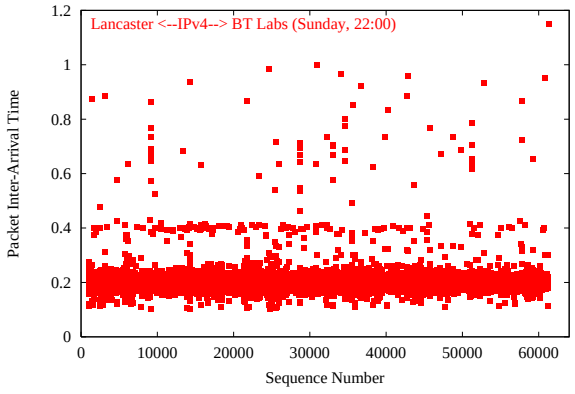


Figure 5.9: Packet Inter-Arrival Times between Lancaster and Ipswich on IPv4

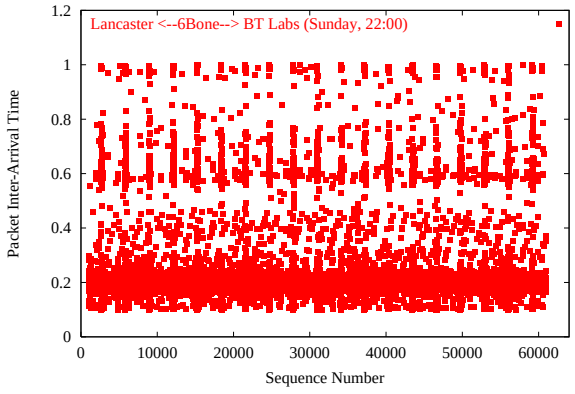


Figure 5.10: Packet Inter-Arrival Times between Lancaster and Ipswich in the 6Bone

The analysis of the packet inter-arrival times lead us to an interesting observation regarding the network QoS within the 6Bone. Figures 5.9 and 5.10 illustrate the packet inter-arrival times recorded during *off peak* experiments between Lancaster and Ipswich on IPv4 networks and the 6Bone. The results of the IPv4 experiments during *off peak* time meet our expectations, whereas the results of the 6Bone experiment look odd at first glance. In periodic intervals (after approximately 3000 packets) the inter-arrival time of a few packets (approximately 245 packets) increases drastically. A reasonable explanation for this behavior is that the delay peaks are a result of periodic processing within the 6Bone edge routers. The peak interval is about 60 seconds. Since IPv4 tunneling is processed in software within 6Bone routers, it is reasonable to assume that the peak delays are a result of processing overload due to periodic router maintenance.

5.1.3 IPv6 Networks

The experiment was then repeated for native IPv6.

Figure 5.11 presents the network configuration of the IPv6 and RSVP testbed at Lancaster University. The core machine, shown in the center, is a Telebit TBC 2000 router. This is, to our knowledge, the first router to have built-in support for RSVP classification based on the IPv6 flow label. Standard PCs running FreeBSD 2.2.5 serve as the end systems. They are equipped with an extended version of the RSVP software release (rel.42a3) from ISI [ISI98]. In order to make use of the IPv6 flow label within RSVP, the RSVP daemon implementation had to be extended. The required modifications are documented in [SDRS98].

The WebAudio server was operating on the machine named **spock**; the client host was the machine called **rsvp2b** (see Figure 5.11).

The graphs of the network QoS measurements within the testbed are not included here. Since the network load in the testbed is nearly zero, and only one router separates the server from the client, the network QoS is expected to be fairly constant and very good. Nevertheless, the presentation of the packet inter-arrival times on a completely unloaded network (see Figure 5.12) gives insight into the jitter caused by process scheduling irregularities in the end stations and a single network router. Further experiments with a pre-loaded network is discussed in section 5.3.

5.1.4 Summary

The experiments presented in this section show that WebAudio operates within different network environments. They also show that the implementation of RTP and, in particular RTCP, within WebAudio enables the application to estimate the network QoS characteristics during streaming sessions. The WebAudio server could easily adapt its operation based on the QoS feedback information by, for example, increasing redundancy or changing audio coding. Although adaptive mechanisms are currently implemented only within

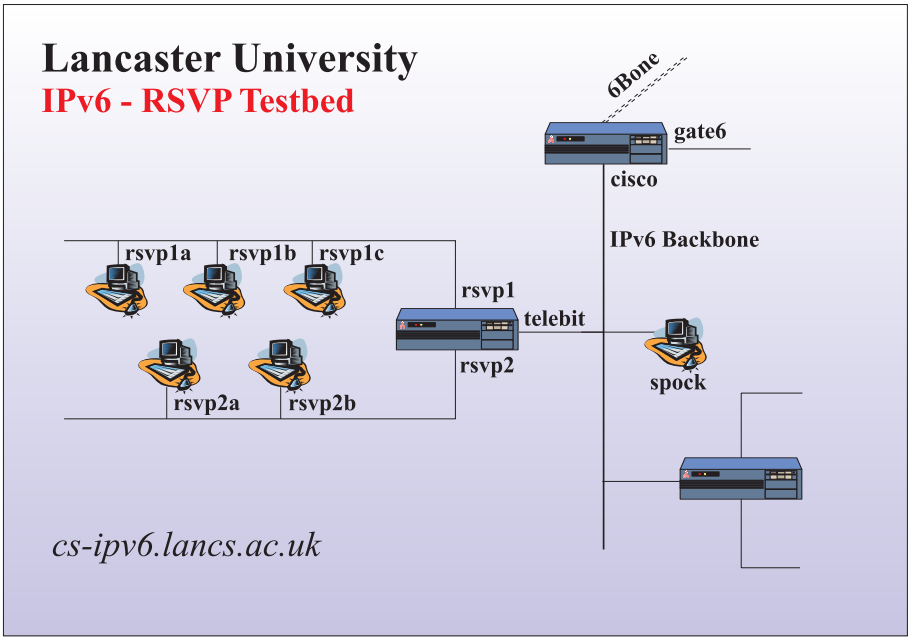


Figure 5.11: The IPv6 and RSVP Testbed at Lancaster University

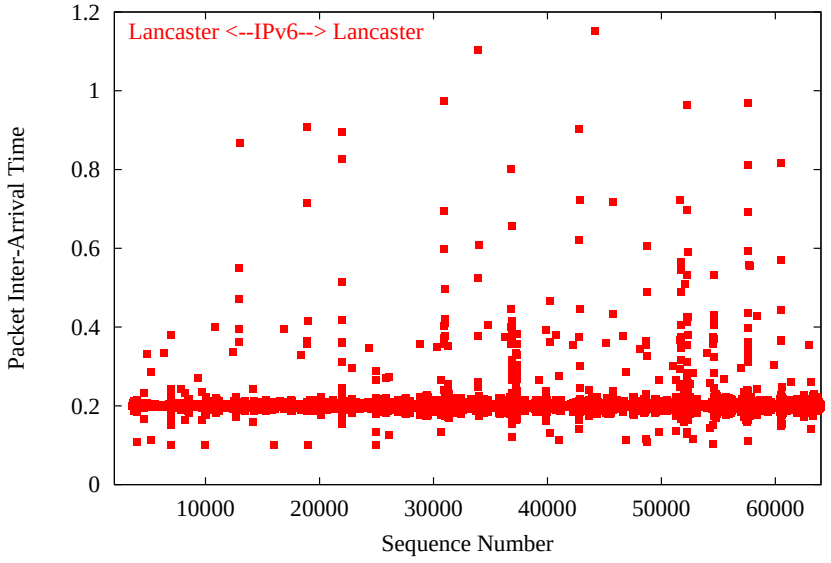


Figure 5.12: Packet Inter-Arrival Times in the local Testbed

the WebAudio client as, for example, the adaptive receiver buffering, server-site adaptation mechanisms are planned in future releases of WebAudio.

Summarizing the results of the network QoS measurements of this section, one can conclude:

- During *off peak* times, when the Internet is not particularly loaded, relatively good network QoS can be achieved (smoothed jitter is often zero; packet loss rate is approximately 0.1%).
- During *peak* times high network load decreases the QoS characteristics of the network significantly (smoothed jitter is mostly non zero; packet loss rate is about 0.4%).
- "Long-distance" Internet communication is not always an indication for bad network QoS. If good connectivity is provided, as in the case of Lancaster and Ulm, in most cases the network QoS during *off peak* is suitable for real-time audio streaming. However, during *peak* times, the QoS degradation within the network is much worse compared to "short-distance" Internet communication.
- Real-time streams in the 6Bone experience odd behavior due to processing bottlenecks in 6Bone routers which process the IPv4 tunneling.

5.2 Resource Reservation

The support for resource reservation within WebAudio can be demonstrated by means of different experiments.

The first approach suggests the comparison of the audio streaming performances on a highly loaded network with and without resource reservations.

The plan for such an experiment in the RSVP testbed (see Figure 5.11) was as follows: as in the experiments presented in the last section, the WebAudio applications record the received network QoS while streaming the audio data from the server (**spock**) to the client (**rsvp1b**). In order to show the effects of resource reservation, the network is overloaded with simple best-effort traffic. This is accomplished by flooding the network[4] from the source **rsvp2b** towards the destination **rsvp1a**. As a result, the Telebit router interface **rsvp1** will be congested since the throughput of the audio stream originating from the WebAudio server **spock** and the throughput of the flooding application exceeds the maximal link bandwidth of the **rsvp1** interface. This network configuration allows applications to be evaluated with varying degrees of resource reservation and network load.

[4]Flooding the network simply means to send as many packets as possible towards a certain destination on the network. A simple network application, called `throughput`, which estimates the maximum throughput between two end nodes, is used.

WebAudio sessions which reserved the required resources before streaming the data are not expected to be affected by the best-effort traffic of the flooding application. WebAudio sessions without resource reservation, however, should perform badly due to the network congestion.

When the experiment was carried out, WebAudio sessions with and without resource reservation have experienced the same QoS characteristics. The first thought was that the reservation establishment failed. Therefore, the proper operation of RSVP was verified by checking the debug files of the RSVP daemon and exploiting the router facility to display established reservations. However, the check validated the proper installation of the resource reservation. After consolidating the router manufacturer, it was clear why the experiments did not achieve the expected results. The disappointing response from the manufacturer was that the QoS scheduling for IntServ is not yet supported.

The second approach to demonstrate the resource reservation support of WebAudio is simply to show the application log traces of the reservation establishment and the reservation state within the router.

Since the reservation protocol operates equally for IPv4 and IPv6, only the latter case is shown here. In particular the setup of a resource reservation based on IPv6 RSVP with flow label support, is presented. The experiment also demonstrates the RSVP extension for flow label support which was developed in the context of this work.

The experiment can be described as follows. The WebAudio server on **spock** sets up an audio session with the client on **rsvp2b**. Before the audio channel is used to carry data, the necessary resources are reserved.

The first debug trace was recorded at the WebAudio server:

```
[...]
CMD  [26/11 01:30:00] setting random flowlabel: 0xC59D3              (1)
RSVP [26/11 01:30:00] local host info: 3ffe:2101:0:980::3            (2)
RSVP [26/11 01:30:00] rsvp instance initialization successful        (3)
RSVP [26/11 01:30:00] dest host info: 3ffe:2101:0:900::3             (4)
RSVP [26/11 01:30:00] rapi\_session: sessionId 1, fd 6               (5)
RSVP [26/11 01:30:00] filterspec: rsvp2b.cs-ipv6.lancs.ac.<fl>809427 (6)
RSVP [26/11 01:30:00] tspec: [T [160K(160K) p=320K m=3.2K M=6.4K]]   (7)
[...]
```

The application first registers the new session with the local RSVP daemon (line 5). The flow specification `TSpec` and the filter specification `FilterSpec` (see section 2.3.1.3) to be used within the PATH messages is passed to the RSVP daemon (line 6 and 7). The flow label, randomly chosen by the WebAudio server and used to label the packets of the audio stream, is passed within the `FilterSpec` to the daemon.

The second debug trace was recorded at the WebAudio client:

```
[...]
RSVP [26/11 01:30:00] local host info: 3ffe:2101:0:900::3              (1)
RSVP [26/11 01:30:00] rsvp instance initialization successful          (2)
RSVP [26/11 01:30:00] rapi\_session: sessionId 1, fd 6                 (3)
[...]
RSVP [26/11 01:30:01] RapiAsyncUpcall event: RAPI\_PATH\_EVENT         (4)
RSVP [26/11 01:30:01] filterCount: 1, flowCount: 1                     (5)
RSVP [26/11 01:30:01] filterspec: 3ffe:2101:0:980::3<fl>809427         (6)
RSVP [26/11 01:30:01] flowspec: [T [160K(160K) p=320K m=3.2K M=6.4K]] (7)
RSVP [26/11 01:30:01] current bandwidth: 0.000000 is set to 0.0        (8)
RSVP [26/11 01:30:01] current slack term (us): 3588448 is set to 0     (9)
RSVP [26/11 01:30:01] RapiAsyncUpcall event: RAPI\_RESV\_CONFIRM      (10)
[...]
```

The client application also registers the new session with the local RSVP daemon (line 3). Line 4 shows the up-call of the RSVP daemon upon receipt of the first PATH message. The client then transforms the received `TSpec` and `FilterSpec` into the `FlowSpec` and `FilterSpec` for the RESV message. These information are passed to the local RSVP daemon when the reservation is setup. The RSVP daemon then sends the periodic RESV messages along the reverse path to the sender. Finally after successful establishment of the reservation, the confirmation is received. Line 10 shows the up-call of the RSVP daemon upon receipt of the RESV_CONFIRM message.

Although the receipt of the RESV_CONFIRM message indicates the successful establishment of the reservation, the reservation state of the Telebit router after the establishment process is also shown. The internal state information of the router validate the proper establishment of the reservation:

```
> show rsvp reservation
session                        filter spec                    flow spec
destination address  port      source address      port        avg   peak
-------------------------------------------------------------------------
3ffe:2101:0:900::3   8888      3ffe:2101:0:980::3  809427      160K  320K
>
```

The reservation `filter spec` contains the flow label of the audio stream rather than the transport port of the source.

Conclusion:
This experiment shows the accurate operation of the resource reservation support within WebAudio although the effects of the reservations regarding the network QoS could not be demonstrated.

5.3 Performance Analysis of Packet Classification

The introduction of the IPv6 flow label in the Internet Protocol header is intended to enable classification of packets according to their destination and service. Reservation protocols, such as RSVP, can make use of this stream identifier to reserve resources for particular streams in the routers along the transport path. This section explores whether there is an efficiency gain due to the use of low level flow labels. A theoretical performance estimate of the different packet classification approaches, namely IPv4, IPv6 and IPv6 with flow label support is presented. These results are validated through measurements from a series of experiments that have been carried out with the WebAudio applications in the context of this thesis.

5.3.1 The Benefits of the Flow Label

According to the IPv6 specification[DH95], the flow label field is intended to label packets that must be classified within intermediate network nodes in order to provide special services, such as non-default QoS or real-time services.

The flow label properties are ideal for proper and efficient packet classification. Three significant characteristics are introduced here. First, the flow label **identifies** packets requiring special treatment. A flow label of zero indicates that a particular packet does not belong to a flow. This allows routers to immediately identify (a simple check within the IPv6 header is sufficient) whether a packet needs special handling or not.[5] Second, the flow label in conjunction with the source IP address serves as a **unique** identifier for flows. This is true because each source node must ensure unique local flow labels according to the IPv6 specification. The great benefit for packet classification is that all the information needed to uniquely classify packets is available within the IPv6 header – where it should be. Third, the flow label is chosen **(pseudo-)randomly** from the uniform space 1h-FFFFFh[6]. The advantage of this attribute is that any set of bits within the flow label field is suitable for use as a lookup-key by routers.

5.3.1.1 The Layer Violation Problem

IPv4 has no implicit support for flows[7]. Thus, intervening routers rely on transport protocol or application level information to identify different flows with the same source. The fact that a router which is supposed to process data only at the network layer, according to the OSI reference model, requires information from the transport or application protocol (i.e. socket ports) to map packets on to their reserved resources, introduces what is

[5] This benefit can only be fully exploited if the flow label is consistently used within RSVP for IPv6.
[6] According to[DH98], the IPng working group agreed to reduce the flow label size to 20 bits.
[7] IPv4 has no equivalent header field to the IPv6 flow label.

known as the *Layer Violation Problem* [SDRS98]. Layer violation has serious drawbacks with respect to the performance of packet classification. Accessing higher layer protocol information to distinguish different flows of the same host pair is an expensive operation, especially in IPv6 networks (see section 5.3.2). Another disadvantage caused by this dependency on transport or application protocol information is that IP-level security techniques, such as IPSEC [Atk95c], cannot be used in conjunction with RSVP. These mechanisms generally encrypt the entire transport header hiding the port numbers of data packets from intermediate routers. In order to overcome this limitation, a "work around" solution has been specified[BO97].

The most important advantage arising from flow label utilization is that it resolves the implicit *Layer Violation Problem* of packet classification. In current RSVP releases packet classification still depends on the transport protocol ports. Section 5.3.2 shows the performance degradation caused by layer violation. Since packets must be classified on a per-packet basis in every intermediate router along a transmission path the performance decrease is significant.

In addition, utilizing the flow label for packet classification has the following advantages:

- Use of the flow label decreases the average processing load of the network routers, and therefore, reduces the end-to-end delays of audio streams: first, when the flow label is consistently used to indicate real-time flows, routers need to perform packet classification only for packets with non-zero flow labels; and second, the processing time of the IPv6 header, especially the extension headers, is greatly reduced due to the fact that all packets from the same flow must have identical extension headers. As a result, routers along a path have to process the headers only on a per flow basis rather than a per packet basis.

- Flow label usage facilitates end-to-end IP-level security mechanisms within resource reservation. As long as packet classification does not rely on higher level information (i.e. ports), IPSEC mechanisms, specifically encryption, cannot obscure important information. This is advantageous for most audio applications since privacy is often an important requirement.

- Reserving resources by means of the flow label has provision to reduce problems caused by frequent route changes such as route *fluttering*[Pax96b]. The flow semantics of IPv6 could be exploited to support route *pinning* mechanisms for reserved flows.

- The flow label has potential to facilitate implementation of QoS based flow routing mechanisms. Ongoing research in the area of label switching suggests approaches which make use of the flow label to identify a packet's reservation state and path [D$^+$98a].

5.3.2 Theoretical Performance Estimate

This section presents a theoretical performance estimate of the classification process as deployed within RSVP. Since no early work could be found which had analyzed the performance of packet classification for RSVP over IPv4, IPv6 and IPv6 with flow label support, a simplified model of the packet classifier process is defined. Based on this simplified model, a complexity comparison between the different packet classification approaches is presented. The model assumes a software processing architecture. Therefore, the results presented in this section hold only for software based packet classification. The processing cost of the different operations might change drastically when performed in hardware. Nevertheless, this analysis gives at least an insight into the complexity of the operations which might be reflected in the production costs of such hardware support.

5.3.2.1 The Model: A Simplified Packet Classifier

A *Packet Classifier* with a network router can be modeled by a process that determines a packet's flow based on the source and/or destination address, and the flow label or some higher level information such as the transport ports. A lookup in the flow state table determines the QoS class of the packet.

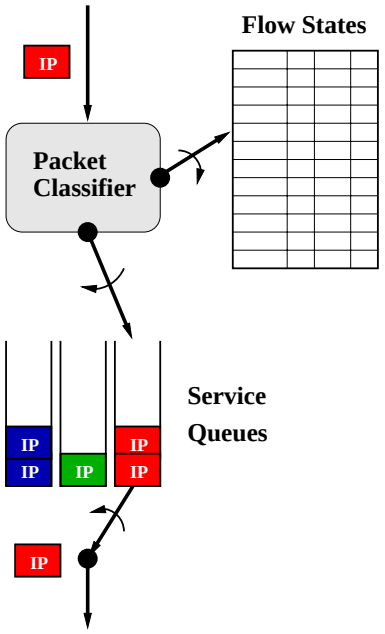


Figure 5.13: *The Simple Packet Classification Machine*

The model — a *Simple Packet Classifier Machine* (see figure 5.13) — is defined according to the RSVP specification [B+97b] and the extension draft for flow label usage [SDRS98].

In general, packet classification within RSVP capable routers is achieved by identifying a packet's *session*[8] and applying the *filters*[9] of the corresponding session.

In the case of standard RSVP (without flow label support), packet classification requires the following steps. First, the classifier must determine whether the packet belongs to a session. Packets that belong to an RSVP session can be identified by (1) looking up the destination IP address in the session table followed by a comparison of the protocol ID and transport port. To minimize the processing cost of the address based table lookup, sophisticated algorithms such as Hashing[10] or Patricia Trees[11] can be used. A packet that matches a session's destination IP address and transport port most likely belongs to the session although the protocol ID has not been compared at this stage. Also, currently deployed RSVP applications use generally UDP as transport protocol. Thus, it seems that packet classification performs better by (2) comparing a packet's transport port against the session table entry before (3) verifying the protocol ID. In a second step, the packet classifier must apply the session filters and determine if a packet is matched. Therefore, (4) the packet's source IP address must be compared with the filter address followed by (5) the transport port comparison.

Packet classification within IPv6 RSVP with flow label support can be processed in a much more efficient manner due to unique and random distribution of flow label values used in the IPv6 header. The flow label can be used as a key for (1) a lookup in the flow state table. Although it is fairly unlikely that different flows would have the same flow label[12], the flow's source IP address is used to resolve such collisions. Thus, (2) a comparison of the source IP address with the address is also necessary.

Based on the classification mechanisms described here, the following operations are defined on the *Simple Packet Classification Machine* (see Table 5.1).

The different structure of IPv4/IPv6 headers and transport protocol headers requests special operations (for example, GetTransHdr) to access the classification information. Row 3 presents an optimistic case where all operations have the same relative processing cost. Row 4 shows more realistic relative costs which are estimated based on a computational analysis. For example, IPv6 address operations (i.e. LookUpAddr, CompAddr) are more expensive than in the case of IPv4 due to the different address size.

Based on the different classification approaches, as described above, and the operations supported by the *Simple Packet Classification Machine*, the per *Packet Processing Cost (PPC)* of either approach can be computed as follows:

[8] A *session* is identified by the IP address, protocol ID and transport port of the destination[B+97b].

[9] *Filters*, defined by filter specs, identify the source, namely the source IP address and the transport port or flow label[B+97b].

[10] A hash key needs to be generated from the IP address; very fast lookup (only one memory access required).

[11] Not very efficient in software due to frequent memory accesses[D+97].

[12] The probability of a key collision with 14 bit key size is $\frac{1}{2^{14}} = \frac{1}{16384}$.

Operation	Description	Relative Processing Cost	
		optimistic	expected
LookUpAddrV4	Flow state look up based on the IP address (i.e. hash lookup, Patricia Tree traversal)	1	2
LookUpAddrV6	See LookUpIPv4Addr; except for IPv6	1	5
LookUpFL	Flow state look up based on the IPv6 flow label	1	1
GetTransHdrV4	Find the begin of the transport protocol header in a IPv4 packet (not constant due to variable length Option Field)	1	1
GetTransHdrV6	See GetTransHdrV4; except for IPv6 packets (more expensive if multiple extension headers present	1	2
CompAddrV4	Compare a pair of IPv4 source/ destination addresses	1	1
CompAddrV6	See CompAddrV4; except for IPv6	1	4
CompPort	Compare transport protocol port	1	1
CompProtId	Compare transport protocol ID	1	1

Table 5.1: Operations of the *Simple Packet Classification Machine*

$$\begin{aligned} PPC_{RSVPv4} &= C(LookUpV4) + P_{destaddr} \times \{C(GetTransHdrV4) + \mu \times \\ &\quad C(CompPort) + P_{sess} \times [C(CompProtId) + C(CompAddrV4) + \\ &\quad P_{filt} \times (\nu \times C(CompPort))]\} \end{aligned} \tag{5.1}$$

$$\begin{aligned} PPC_{RSVPv6} &= C(LookUpV6) + P_{destaddr} \times \{C(GetTransHdrV6) + \mu \times \\ &\quad C(CompPort) + P_{sess} \times [C(CompProtId) + C(CompAddrV6) + \\ &\quad P_{filt} \times (\nu \times C(CompPort))]\} \end{aligned} \tag{5.2}$$

$$PPC_{RSVPv6FL} = C(LookUpFL) + P_{flow} \times \{\sigma \times C(CompAddrV6)\} \tag{5.3}$$

The *Cost* function $C(x)$ determines the processing cost of operation x. Table 5.1 (row 3 & 4) specifies the relative processing cost of each operation when performed on the *Simple Packet Classification Machine*.

The *PPC functions* are based on the implicit assumption that packets belonging to a flow use the same protocol ID (for example, UDP). Therefore, if a packet's destination IP

address and port matches a session, the packet belongs to the corresponding session. This simplification does not restrict universality. Also, to simplify flow label based classification, it is assumed that all 20 bits of the flow label are used as lookup-key.

In the first analysis (see row 3 of Table 5.1), the symbolic cost $C(x) = 1$ is used for all operations. The results of this analysis represent an optimistic case. Only the steps absolutely necessary for packet classification are considered. In the second analysis (see row 4), more realistic relative processing costs for each machine operation are assigned.

Only a certain percentage of the packets passing an Internet router belong to a flow, and hence, complete classification processing must be performed. In order to take this into account, different probabilities are assigned for packets belonging to a *session* (P_{sess}), being matched by a session's *filter* (P_{filt}) and having the same destination IP address as another registered session ($P_{destaddr}$). The conditional probabilities P_{sess} and P_{filt} are defined as follows:

$$P_{sess} = (Prob(\textbf{packet\_belongs\_to\_a\_session}) \mid P_{destaddr}) \tag{5.4}$$

$$P_{filt} = (Prob(\textbf{packet\_is\_machted\_by\_any\_filter}) \mid P_{sess}) \tag{5.5}$$

In the case of flow label based classification, P_{flow} is used as the probability for a packet belonging to a flow. The relationship between P_{flow} and the probabilities in the other approach is described here:

$$P_{flow} = P_{dest} \times P_{sess} \times P_{filt} \tag{5.6}$$

The factors μ, ν, and σ take into account that certain operations might have to be performed multiple times in order to resolve ambiguities. The factor μ determines how often a destination transport port comparison must be performed on average. Assuming that on average 1.5 RSVP sessions are active between a given host pair where an RSVP session is active, μ equates to 1.25. The factor ν indicates how often, on average, a source transport port comparison must be performed. The result is 1.5 if one assumes that on average two flows are active between the communicating hosts. Factor σ specifies how often the IPv6 address must be compared on average due to lookup collisions caused by identical flow labels being chosen by different hosts. Since this is fairly unlikely, as discussed above, the value is in a first analysis set to 1.01.

Figure 5.14 and 5.15 present the graphs of the *PPC* for either analysis, namely the optimistic and the likely realistic case, by varying the probability $P_{destaddr}$ that a packet's destination IP address matches any session IP address.

Analysis 1 clearly shows the great advantage of deploying the flow label within RSVP over IPv6. The *PPC* increases much slower due to the fractional dependency between P_{flow} and

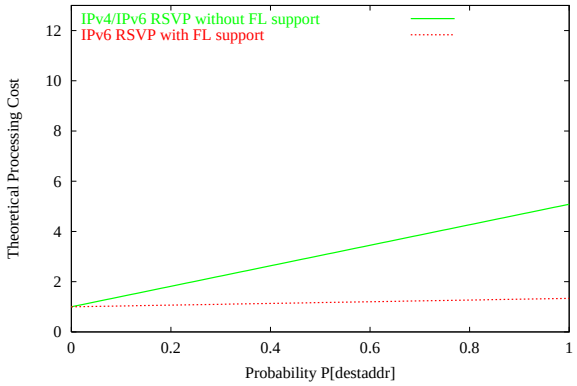


Figure 5.14: Analysis 1 – Optimistic-case *PPC* for variable $P_{destaddr}$ (all classifier operations have a symbolic cost of 1)

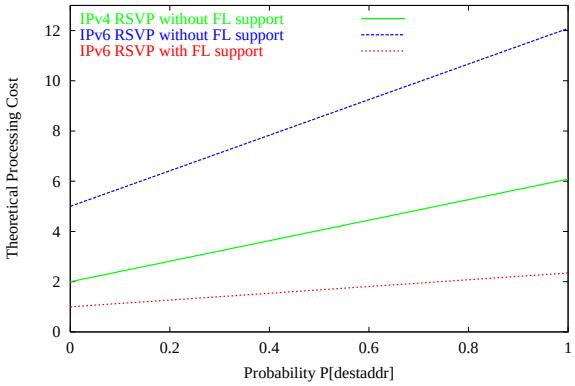


Figure 5.15: Analysis 2 – *PPC* for likely realistic relative operation costs (row 4 of Table 5.1)

$P_{destaddr}$ as shown in (6) and especially due to the simpler processing. Even in the optimistic case, where all operations have a minimal cost of 1, the flow label based approach performs much better. The explanation is simple. First, the flow label, used as lookup-key, enables a quick check as to whether the packet belongs to a flow or not. Additional classification processing cost is incurred only if a packet belongs to a flow, whereas for standard RSVP processing, it is already costly to find out if a packet has to be classified or not. Second, besides the flow table lookup, only the source IP address needs to be compared in order to identify the proper flow, whereas for standard RSVP classification, the protocol ID comparison as well as the layer violating transport port validation are required. Analysis 2 (Figure 5.15) presents a more likely and realistic progression of the *PPC* function due to the adjustment of the operation costs (see row 4 in Table 5.1).

In order to examine how stable our model is, an additional analysis is presented. Figure 5.16 shows the progression of the *PPC* functions for other values of μ, ν, and σ. Here, it is assumed that on average 3 RSVP sessions are active ($\Rightarrow P_{sess} = 0.33$) and 5 flows are in use ($\Rightarrow P_{filt} = 0.2$). Therefore, the value of μ results to $1 + \frac{2}{3}(1 + \frac{1}{3} \times 1) = 1.89$, whereas ν results to $1 + \frac{4}{5}(1 + \frac{3}{5}(1 + \frac{2}{5}(1 + \frac{1}{5} \times 1))) = 2.51$. It is also assumed that the probability for a flow label collision is 10%. Thus, σ results to 1.10. The graph clearly shows that the change of the average number of sessions and flows between a given RSVP host pair and the modification of the flow label collision probability does not change the former results of the *PPC* much. This suggests that the results are fairly stable.

Figure 5.17 aims to illustrate the impact of flow label collisions on the classification performance. A 3-D graph is shown where the X axis determines the $P_{destaddr}$, the Y axis determines the maximum number of flows with an equal flow label (Ψ), and the Z axis represents the *PPC*. Since the expected number of source IP address comparisons σ and the probability that a packet belongs to a flow P_{flow} is closely related, the formula 5.3 is enhanced to:

$$PPC_{RSVPv6FL} = C(LookUpFL) + P_{flow} \times \{(P_{flow} \times \Psi) \times C(CompAddrV6)\} \quad (5.7)$$

The graph shows that even with a relatively high number of average flow label collisions $(P_{flow} \times \Psi) - 1$, the flow label based classification approach excels standard RSVP classification.

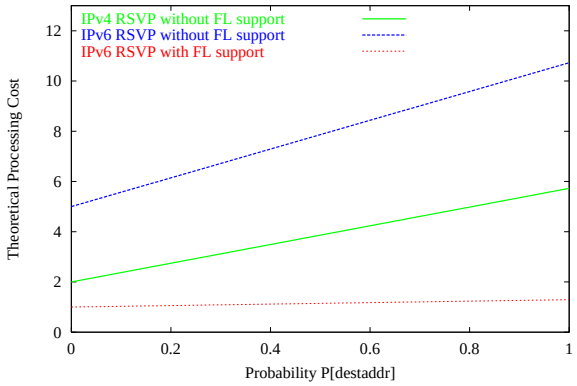


Figure 5.16: Analysis 3 – *PPC* in the likely realistic case but with 3 sessions, 5 flows and a flow label collision probability of 10% on average

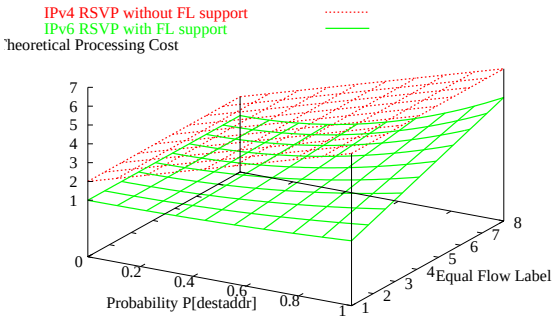


Figure 5.17: *PPC*, computed from $P_{destaddr}$ (X axis) and Ψ (Y axis) according to 5.7, shows that flow label collisions have a minor impact

To summarize this theoretical analysis shows that classification based on the IPv6 flow label has the potential to outperform standard RSVP based classification for IPv6 by far (about 3-6 times) and even the IPv4 classification (about 2-4 times) although the latter should be much simpler due to shorter IP addresses and the simpler *Option* mechanism. Furthermore, it has illustrated that the problem of flow label collisions, which should occur rarely if lookup tables are sufficient in size, has only a minor impact on the classification performance.

5.3.3 Experiments

This section confirms the results of the theoretical analysis through experiments with WebAudio in a real network environment.

To show that the processing load of a router is not as strongly affected by the number of flows when classification is based on the flow label, one can measure either the maximum number of packets per second (or flows) that can be processed by the router or the processing load while transmitting the same number of packets per second for each approach. The latter measurement technique is chosen within this work since the former approach might lead to incorrect data as a result of operating the router at its processing limits.

The RSVP testbed used for the experiments is shown in Figure 5.11. The TBC 2000 router has three interfaces, named *rsvp1*, *rsvp2*, and *telebit*. Each interface has an independent processor to perform packet classification and scheduling. The processing load of each individual interface can be determined with the *measure* command offered by the router's operating system. This feature allows measurement of the load on the interfaces caused by classification of packet audio as required by RSVP. The WebAudio application is used for audio streaming. Resource reservation for the audio streams is accomplished through our modified RSVP daemon which supports resource reservation for standard IPv4/IPv6 flows and marks flows with the IPv6 flow label.

Based on this setup, several measurements were made to compare the performance of the different classification implementations, namely IPv4, IPv6 and IPv6 with flow label support. Figure 5.18 presents the results of average experiments measuring the per *Interface Processing Load (IPL)* of the router while periodically establishing new audio streams and their RSVP reservations. Each experiment has a duration of about 15 minutes. At the end of every 30 second period a new audio stream was setup. An additional reservation taking into account the QoS demands of the new audio stream, is established. The audio streams, which all transmit the same audio data, require a bandwidth of 200 Kbps. Every 50 ms, an audio frame (1250 bytes) with its RTP header encapsulated in a UDP packet is sent independently for each flow. The *measure* command offered by the TBC 2000 is used to determine the progression of the *IPL* throughout the experiment. Each time the *measure* command is called, it prints the average processing load since the last call for each interface. As a result, by measuring the load every minute while steadily increasing the number of flows, the load increase of the interfaces can by closely traced.

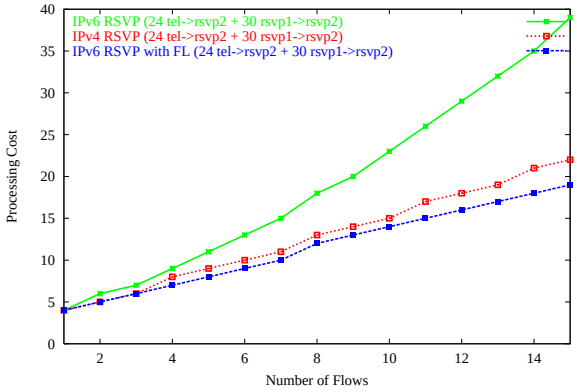


Figure 5.18: Progression of the *Interface Processing Load* while increasing the number of audio flows

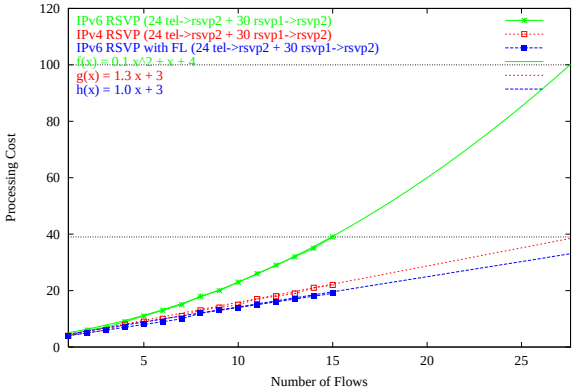


Figure 5.19: Extrapolation of the *Interface Processing Load* progression

The experiments, presented in Figure 5.18, are based on the following setup: **rsvp1a** and **spock**, transmit their audio data to **rsvp2b** and **rspv2a** respectively. Therefore, both hosts send their packets through the router interface **rsvp2**. As a result, it is expected that the load on that interface increases heavily due to the required QoS processing,

namely packet classification and scheduling. Running each experiment for all three classification approaches, allows a direct comparison of the load measurements. Note that packet scheduling is absolutely independent of the packet content and hence, not affected by the flow label. The results of establishing 1-24 flows between **spock** and **rsvp2a** and 1-30 flows between **rsvp1a** and **rsvp2b** clearly show the advantage of using the flow label for packet classification.

These results, namely the flow label based classification performs best, followed by the standard IPv4 and finally the IPv6 approach, are consistent with the theoretical analysis presented in section 5.3.2. By extrapolating the graphs (Figure 5.19), one can see the extent to which the flow label approach performs better. While the standard IPv6 approach might reach maximum capacity after approximately 28 minutes (56 flows), flow label based classification can deal with about 3 times this level (168 flows) before reaching the router's processing limit. Comparing the load progressions with our theoretical results shows that the difference between the approaches, and especially between the flow label based classification and the IPv4 approach, is less than expected.

This first experiment is not very realistic due to the lack of standard *best-effort* traffic competing with the traffic of the reserved flows in the router. Therefore, in a second trial a more realistic experiment is set up by adding a constant *best-effort* load of 2 Mbps between both host pairs. Figure 5.20 presents the results of this extended experiment. Note, the final *IPL* of the standard IPv6 based classification experiment is already above 65% of the maximum capacity whereas the classification based on the flow label requires only about 30%. The former requires about 50% of the maximum capacity for the flow processing whereas the latter consumes only 10%. Figure 5.19 extrapolates the progression of the processing load of this second trial in order to compare the results with our first experiment. After 28 minutes (56 flows), in the case of standard IPv6 classification, the processing capacity would already be overloaded by approximately 20% whereas flow label based classification loads the router only to approximately 40%.

The results of the second experiment are in accordance with the findings of our theoretical analysis. The performance ratio of the theoretical classification analysis results to 1:3:5 for flow label based, IPv4 and IPv6 classification whereas the experiments discover a ratio of 1:1.4:4. The theoretical analysis takes only the processing cost of classification into account. The results of the experiments, however, represent the performance difference of the overall *IPL* which accumulates the load of packet classification and scheduling. Especially in the case of IPv6 flows, the *IPL* is higher due to bigger IP headers which must be processed. Considering this, it is surprising how closely related the results are. From this, one can conclude that packet classification has a relatively strong impact on the overall *IPL* within the TBC 2000. This might also be true for other router brands.

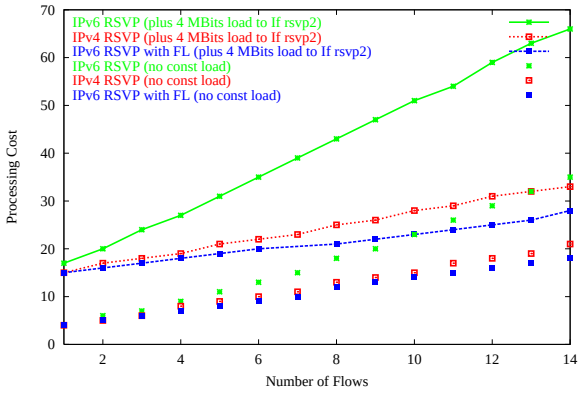


Figure 5.20: Progression of the *Interface Processing Load* while increasing the number of audio flows with a constant base load of 4 Mbps on the interface

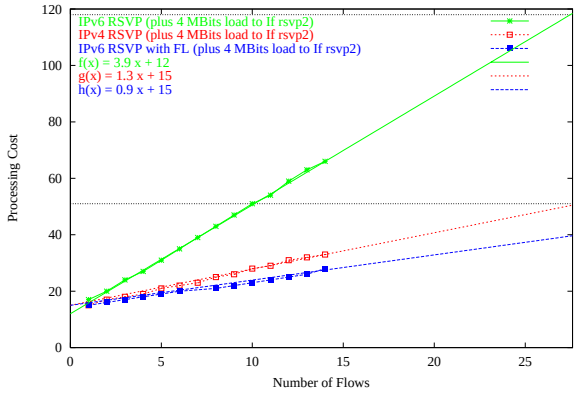


Figure 5.21: Extrapolation of the *Interface Processing Load* progression

5.3.4 Summary

Experiments on early commercial IPv6 routers have shown that flow label based packet classification within resource reservation protocols, such as RSVP, decreases the processing costs in routers of the order of 3-6 times with respect to standard IPv6 classification and about 2-4 times with regard to IPv4 classification. The results vary depending on the available processing architecture. As a result, utilization of the flow label within resource reservation enables routers to handle significantly more flows before reaching their processing limit and reduces the end-to-end delay due to simpler and hence, faster packet processing in each intermediate router along the transmission path.

Conclusion:
The results of the theoretical analysis, presented in section 5.3.2, are confirmed by the experiments. Both show the same results: flow label based classification performs significantly better than standard IPv4 classification and greatly outperforms standard IPv6 classification.

5.4 Summary

The experiments presented throughout this chapter can be divided into three groups:

1. Experiments within different network environments, namely IPv4, IPv6 and the 6Bone.

   The results of these experiments can be summarized by stating that:

- WebAudio operates well within different network environments.
- The QoS feedback mechanisms within WebAudio enables the application to estimate the network QoS characteristics during streaming sessions.

2. Experiments with the resource reservation protocol RSVP.

   The experiments validate the accurate functioning of resource reservation signalling within WebAudio. The effects of the reservations regarding the network QoS could not be demonstrated due to lack of support for QoS scheduling in the router.

3. Experiments to verify the performance gain of packet classification based on IPv6 flow labels.

   The results can be outlined as follows:

   - The flow label-enabled and extended RSVP implementation developed in the context of this work operates correctly.
   - Flow label based packet classification decreases the processing costs in routers of the order of 2-4 times compared to IPv4 classification, and it outperforms standard IPv6 classification about 3-6 times.

Chapter 6

Final Remarks

The final chapter of this thesis concludes the work by summarizing the achievements presented throughout the thesis. The last section of this chapter provides an outlook on further developments on WebAudio and future research.

6.1 Conclusion

The main achievements of this work can be divided into three parts.

First, the thesis has provided an in-depth discussion on important Internet multimedia protocols and mechanisms to improve the QoS of real-time audio streaming in the Internet. The results of this study laid down the foundation for many design choices during the development of WebAudio.

Second, WebAudio, a state-of-the-art real-time audio streaming application, has been developed and implemented. This application relies on several important protocols in the context of Internet multimedia and real-time streaming. Based on the novel application WebAudio, further investigations regarding the impact of the IPv6 flow label on resource reservation mechanisms, such as RSVP, were carried out.

Third, the results of a theoretical analysis to quantify the performance gain introduced by the use of the flow label for packet classification as required within RSVP have been presented. These results were confirmed by performance measurements within an RSVP testbed.

The following sections summarize the results of the different areas of investigation along with their conclusions.

6.1.1 Study of Internet Protocols and QoS Mechanism

Interactive real-time audio data communications are time-critical with stringent QoS constraints regarding the end-to-end delay, jitter and reliability.

Chapter 2 and 3 provide an extensive discussion on important Internet multimedia protocols and mechanisms to improve the QoS of interactive real-time streaming in the Internet.

While the first part of this section presents the conclusions of the Internet multimedia protocol study, the second part concludes the analysis of current QoS mechanisms with respect to interactive media streaming as presented in chapter 3.

Conclusions on Internet Protocol Study

The most important protocols currently used within Internet multimedia applications were reviewed. The study explores their usability for interactive real-time streaming in the Internet. The advantages and drawbacks of these protocols were discussed following the structure of the OSI reference model.

The network layer protocols IPv4 and IPv6 were discussed; the latter was emphasized. The benefits of the new Internet Protocol regarding real-time media streaming can be summarized as follows: first, IPv6 extends the protocol by native multicast and security facilities, second, simplification of the IPv6 protocol header enables faster processing within intermediate network nodes, and finally, the IPv6 flow label resolves the implicit layer violation problem of RSVP and improves the performance of packet classification.

The transport protocols and streaming mechanisms discussed were: UDP, TCP and RTP-on-UDP. Whereas TCP is not suitable for interactive real-time streaming applications because of the interference of its congestion control and reliability mechanisms with the requirements of time-critical applications, UDP provides a simple but sufficient service. Control mechanisms such as packet ordering, for example, need to be implemented on higher levels. As a result, RTP-on-UDP is recommended for real-time streaming, since it adds valuable stream information to the packets and provides a feedback mechanism by means of its control protocol RTCP. RTP, on its own, is misleadingly called a transport protocol. It is merely a lower-level application support protocol.

The resource reservation protocols RSVP and YESSIR were discussed. RSVP is currently the resource reservation protocol of choice within the IETF. It is recommended to be used with real-time streaming applications to negotiate *guaranteed* or *controlled load* end-to-end QoS if the IntServ architecture is supported along the network path. The network layer reservation protocol has, however, the following drawbacks: first, it is not scalable within the core of the network where thousands of flows need to be maintained, and secondly, RSVP is not a reliable reservation protocol because it follows the dynamics of the underlying routing protocol. The simple and light-weight reservation protocol YESSIR,

in contrast, is not a network-level signalling protocol. The signalling is simply based on top of the application-level protocol RTP (RTCP) and hence can only provide very limited reservation services.

Regarding application level protocols HTTP and RTSP were discussed and evaluated for the use of stream control. HTTP as a standard protocol can only provide simple stream control; the lack of session or stream semantics limits its usability. RTSP, in contrast, is a sophisticated stream control protocol with similar functionality to a "VCR remote control". Whereas RTSP-based stream control requires special RTSP capable clients, HTTP-based stream control can be achieved with standard Web browsers. Due to the similarity of their protocol designs, RTSP could be easily extended to provide service for the simple HTTP control mechanism transparently.

Conclusions on QoS Mechanism Study

An analysis of application level techniques and network level QoS mechanisms that have an impact on the QoS of interactive real-time media streaming applications were presented. The results are summarized below.

Application layer mechanisms, namely packet transfer, forward error correction, adaptation and receiver buffering were examined regarding their usability and importance for interactive real-time streaming:

With respect to packet transfer, interactive real-time streaming applications have a choice of transport mechanisms, packet sizes and packet transmission techniques. RTP-on-UDP is recommended for the packet transfer. End-to-end delay constraints require interactive applications to minimize packetization delays by using small packets. A payload of one or two media frames is suggested. Non-interactive application should maximize the number of frames per packet in order to minimize the packet overhead. Moreover, interactive real-time media streaming applications are advised to "shape" their data traffic so that packets are sent isochronously over time to reduce the likelihood of packet clustering and thus the packet loss rate.

The use of packet-based forward error correction mechanisms capable of correcting a few consecutive packet losses is recommended within interactive real-time streaming. Since the number of consecutive packets lost is usually small (of the order of 1 to 3 packets), these FEC mechanisms are effective in the Internet.

Adaptation as a general mechanism for adjusting the operation of an application depending on the QoS provided by the network is very effective for Internet real-time streaming. Even quite severe service fluctuations can be accommodated by means of adaptive mechanisms. However, adaptation can only cope with QoS degradations of deterministic bounds. Since resource reservation mechanisms are not supported in most parts of the Internet yet, application-level adaptation is a necessity for real-time streaming.

Receiver buffering, for example, benefits from adaptive mechanisms. In general, adaptive (dynamic) receiver buffering is preferable over simple (static) buffering since "optimal" buffering delay estimation depends highly on the jitter dynamics. If no QoS guarantees are granted, receiver buffering is mandatory in order to compensate for the jitter. To compensate for the jitter introduced by process scheduling in the receiver node, the buffers of the sound device need to be exploited. Again adaptive buffering is preferable since the buffering delay can be minimized.

Different **network layer mechanisms** to improve the QoS of real-time streaming applications or to guarantee QoS for these applications were discussed.

The first group of mechanisms includes various service differentiation mechanisms: relative priority marking, service marking, and the mechanisms proposed to support DiffServ.

The simple approach of relative priority marking is not very useful for real-time streaming since relative priorities do not provide suitable differentiation for real-time media streams. Service marking improves the relative priority marking by increasing the range of possible service semantics. However, new service types cannot be easily introduced, since the header field is small and new types would involve changes in every network node.

Differentiated Services is currently the preferred differentiation mechanism discussed within the IETF. It outperforms the two former differentiation mechanisms by supporting flexible service classes which are not limited to a pre-defined standard. DiffServ divides the network into several virtual *best-effort* networks. The scalable architecture has the potential to resolve the scalability problem of the IntServ architecture in the core of the network since no "per-flow" state information is required. However, since DiffServ provides forwarding behaviors only on a "per-hop" basis, it cannot offer end-to-end service guarantees. The lack of a reliable admission control mechanism impedes DiffServ from offering reliable resource promises. Even though DiffServ cannot guarantee end-to-end QoS, when widely deployed in the Internet, it has the potential to significantly improve the network QoS received by current real-time streaming applications. QoS sensitive real-time media traffic would then be protected from *(discrete) data traffic*.

A different QoS mechanism, called IP label switching, aims at improving current QoS in the Internet by means of packet switching techniques. IP label switching is more efficient than regular IP routing due to the simpler decision making process in the network routers (or switches). This has the potential to reduce the end-to-end delay received by real-time streams by reducing the processing cost in every intermediate node. However, IP label switching does not scale well in the core of the network, where IP switches have to maintain forwarding state for every flow.

The IntServ architecture provides network level QoS by controlling the network delivery service. Real-time streaming applications can request their QoS demands by means of a resource reservation protocol. IntServ supports guaranteed (hard) and controlled load (soft) QoS guarantees that provide optimal service for QoS sensitive applications. Although controlled load QoS, which provides service equivalent to unloaded networks, is suitable

for real-time streaming applications, these should still implement adaptive mechanisms to deal with small variations in the QoS. In contrast, guaranteed QoS, which offers hard QoS guarantees, provides an optimal service for real-time streaming applications even without adaptation, error correction, and receiver buffering mechanisms. The main drawbacks of IntServ are: first, IntServ relies on support within all network elements along the transmission path in order to enable end-to-end reservations, and secondly, "per-flow" state is required in every intermediate network element. This does not scale successfully in large networks and in particular not within the core of the Internet. Inter-operation between IntServ and IP label switching is being studied. The state information required in network nodes to perform label switching and resource reservation can be easily maintained together.

Sometimes, IntServ and DiffServ are considered as competing technologies. A clear trend towards IntServ or DiffServ, however, cannot be seen yet and might never be seen. Some researchers believe that IntServ is a "dead" technology due to its scalability problems in the core of the network. However, new approaches or approaches that integrate the IntServ and DiffServ architectures are more likely to become the future QoS framework within the Internet. Integration of DiffServ and IntServ suggests the use of DiffServ as a scalable, hop-by-hop QoS mechanism in the core of the network, and IntServ in the stub networks where scalability is not a problem. Using IntServ as "customer" of DiffServ enables streaming applications to negotiate their QoS requirements within stub networks and allows admission control for the DiffServ region. In the core, IntServ QoS reservations must be mapped to appropriate DiffServ service classes. However, since DiffServ provides only service based on per-hop behaviors, it cannot easily provide real QoS guarantees in the core. Therefore, extended IntServ solutions that achieve scalability due to aggregation and overhead reduction might be preferred in the long term.

Final Results

Since the end-to-end QoS constraints are very strict in the case of interactive real-time audio, QoS degradation has a great impact on the usability (in terms of user satisfaction). In order to satisfy the QoS requirements of audio streaming applications, it is suggested that these applications are adaptive to the "best" QoS mechanism available along the network transmission path. As a result, if IntServ/RSVP is supported, the applications should first make use of this resource reservation mechanism. Otherwise, they should fall back and exploit the service differentiation approach of DiffServ. If neither QoS mechanism is provided by the network, applications have to cope with the simple *best-effort* service. Soft QoS guarantees, as offered by the IntServ controlled load service class, are sufficient for adaptive audio streaming applications. Non-adaptive applications, in contrast, either require hard QoS guarantees or provide non-optimal service due to over-provisioning.

In addition, current real-time audio streaming applications should be developed considering the following design recommendations:

- UDP should be used as transport protocol.
- RTP is a useful application level protocol that facilitates media streaming. In conjunction with its feedback mechanism, RTP and RTCP enable adaptive mechanisms based on QoS feedback.
- RTSP is a comprehensive stream control protocol. HTTP is sufficient for simple stream control. Both protocols can easily co-operate within a multi-protocol interface.
- Interactive real-time streaming requires small packets (1 or 2 media frames) even if the packet overhead is big.
- Receiver buffering is mandatory if no hard QoS guarantees are granted. Otherwise, adaptive receiver buffering is suggested in order to minimize the buffering delay.
- Adaptation is suggested wherever applicable as a general means to deal with QoS dynamics in the network or the end systems.
- Packet-based forward error correction, which corrects up to a few consecutive packet losses, is recommended within Internet streaming.

6.1.2 Real-Time Audio Streaming Application

The real-time streaming application called WebAudio was developed within the context of this work.

The final conclusions of the Internet protocol analysis and the QoS mechanism study presented in the last section had an important impact on the design choices and implementation of WebAudio. The protocols and mechanisms which are considered mandatory and most of the ones which are recommended are included.

WebAudio, yet another real-time audio streaming application, was developed and implemented from scratch for the following reasons: (1) The lack of streaming applications using IPv6 would have prevented the study on the impact of the flow label on network QoS mechanisms such as IntServ/RSVP. (2) The integration, at an early design stage, of the mechanisms and protocols suggested above led to a more efficient design than their addition to an existing application. (3) Available applications support either adaptation or resource reservation, while both mechanisms are suggested for current Internet streaming applications. (4) The demand for a streaming application that can be used within WWW applications in concern with the fact that available streaming applications are not easily or thoroughly Web integrateable suggested a specially designed application.

WebAudio, the live audio streaming application, provides audio conferencing services for multiple users. The client and server application provide an "open" stream control interface which enables easy Web integration. Resource reservation and adaptation mechanisms are exploited to improve the QoS of the audio streams.

The operation of WebAudio can by summarized as follows: The user interface, which is invoked by the Web browser, either as a plug-in or as a helper application, controls the client application by means of HTTP or RTSP. WebAudio uses RTSP for stream control between a client and server.

The main contributions achieved with WebAudio are:

- WebAudio is designed in a platform independent manner. Besides the current release for FreeBSD and Linux, a future version for the Microsoft Windows systems is planned.
- Support for multiple audio encodings is provided. Although currently only PCM and GSM codecs are included, the application design has provision to easily integrate further codecs
- WebAudio operates within IPv4 and IPv6 networks. The network protocol for the stream data can be chosen during application initialization or even run time.
- Network level QoS according to the IntServ architecture is supported by WebAudio. The resource reservation protocol RSVP is used to set up and control the reservations. This enables WebAudio to provide high quality service to end users of IntServ capable networks.
- The packet streaming mechanism provided by RTP-on-UDP is used for the audio stream. RTCP enables the client application to adapt its operation to QoS changes in the network.
- WebAudio uses an adaptive receiver buffering mechanism to compensate for jitter. An extended algorithm for the estimation of the optimal buffering time, which specially considers delay spikes in packet transmission and adapts adequately fast to changes in the network, provides good service.
- The WebAudio client and server are multi-stream capable. Whereas the server transmits the encoded audio data to all the clients on individual point-to-point connections, the clients mix all received streams prior to playback.
- The multi-protocol control interface enables clients to control the applications by means of HTTP or RTSP. The HTTP-based control interface facilitates simple control, whereas the RTSP-based control interface enables the full range of stream control.

- The "open" control interface has provision for simple and flexible user interface development and enables easy Web integration. Even a standard Web browser is sufficient to control the application.

Although the WebAudio client and server applications are still in an alpha release version, the experiments in chapter 5 have shown that they operate well within different network environments and support resource reservation according to RSVP. WebAudio has also been very useful for experimental purposes. In conjunction with the extended IPv6 RSVP implementation, it has been used to verify the theoretical results about the performance gain of IPv6 flow labels (see section 5.3.2) by means of real experiments (see section 5.3.3).

6.1.3 Deployment of Flow Label within RSVP

An important aim of this work was to explore the benefits of IPv6 for network QoS mechanisms based on resource reservation. The thesis examines whether there is an efficiency gain due to the employment of network level flow labels within IntServ/RSVP.

In order to investigate the impact of IPv6 flow labels, the RSVP daemon implementation of ISI [ISI98] had to be extended. The required extensions and modifications were documented in an Internet Draft [SDRS98].

Based on the enhanced RSVP implementation and WebAudio, a series of experiments was performed to measure the performance gain of packet classification when the IPv6 flow label is used as primary classification criterion. The results of these measurements can be summarized as follows: Flow label based packet classification decreases the processing costs in routers of the order of 3-6 times with respect to standard IPv6 classification and about 2-4 times with respect to IPv4 classification. The results may vary depending on the available router architecture. As a result, utilization of the flow label within resource reservation enables routers to handle significantly more flows before reaching their processing limits. Flow label use within IntServ/RSVP has also the potential to reduce the end-to-end delays of packets due to simpler, and hence faster, packet processing in each intermediate router along the transmission path.

Besides the performance gain of packet processing within network nodes, the flow concept of IPv6 contributes also on an other level. It resolves the implicit *Layer Violation Problem* of standard RSVP and enables IP-level security mechanisms without recourse to "clandestine" techniques.

Finally, from the flow label examination one can conclude that IPv6 has the potential to improve real-time audio streaming applications due to simpler, and hence substantially faster, processing in the network routers.

6.2 Future Work

An overview of future plans and development work concludes the work of this thesis. The first section describes the further developments within the WebAudio application. The second part discusses future research following from this work.

6.2.1 Further Development

Chapter 4 identifies various areas where future developments within the WebAudio applications are planned.

Enhancements within WebAudio are aimed primarily at the following directions ordered according to their importance:

First, a graphical, Web-based user interface for control of the WebAudio client application will be developed. The HTTP-based "open" control interface allows a simple, but effective, design based on HTML and Java Script.

Second, WebAudio will be ported to the Microsoft Windows operating systems, namely Windows 95 and 98. A version for Windows NT may be developed later. This will allow WebAudio to be a freely available, state-of-the-art real-time audio streaming tool which can be widely deployed in the Internet. Since the application was carefully designed, keeping in mind the aim of platform independence, the port to Windows 95 or 98 is expected to be easy to accomplish.

Third, several new audio codecs will be included in WebAudio in order to cover the full range from low-bandwidth voice to high-quality sound codecs. Again these changes are expected to be fairly simple since the *Audio* class has already provision to support additional audio encodings. Support for multiple audio formats is especially simple on Windows systems, since the distribution already contains the most common codecs.

Fourth, support for the DiffServ architecture will be integrated. Since DiffServ is distinct from IntServ, the main alternative QoS mechanism currently discussed within the IETF, DiffServ support within WebAudio would enable the application to adaptively select the "best" QoS mechanism provided in the network. However, it is not yet clear how network QoS will be achieved within the future Internet.

Fifth, multicast based audio streaming will be considered in future versions of WebAudio. Since the server is currently limited to send the audio data in form of multiple point-to-point streams, the number of clients is restricted to small and moderate groups. The extensions for multicast support are expected to be small, since the current protocols used within WebAudio are aware of multicast communication.

Sixth, packet-based FEC mechanisms that compensate for a few consecutive packet losses will be integrated within WebAudio. However, since the suggested FEC mechanism has

an impact on adaptive buffering delay estimation, this change might require further investigations in these areas.

6.2.2 Further Research

Finally, future experiments and research which could not have been completed within the framework of this thesis, are noted here.

WebAudio encompasses an enhanced version of the adaptive receiver buffering algorithm from UMASS [MKT98] which improves the algorithm responsiveness and decreases its complexity. However, although the arguments for the improvement seem reasonable, it has not yet been proven on large-scale experiments within Internet real-time streaming. Performing these experiments is one future topic of research.

WebAudio uses the resource reservation protocol RSVP. With respect to the discussion in section 2.3.1, RSVP has still significant problems if used in large-scale networks such as the Internet. The drawbacks addressed within this context are as follows. First, RSVP has significant protocol overhead and scalability problems within core network routers due to the periodic, "per-flow" messages. In [MHSC99], it is suggested to replace the "per-flow" soft-state with a "per-neighbor" soft-state and use hard-state for the flows. RSVP processes within network routers make sure that their RSVP neighbors operate properly by means of so-called "heartbeats". Therefore, when a flow reservation is set up, and the neighbors are "alive", the periodic messages for the flow reservations can be renounced. Secondly, RSVP has very slow reservation establishment times in environments where packet loss is high. If the initial PATH or RESV message is lost, the message is not re-sent before the refresh period times out. A fast establishment mechanism is suggested [MSH98, MHSC99] which re-sends the initial PATH message after a very short timeout period (which backs off) until it receives the RESV message as confirmation for the successful PATH message. Further developments and experiments regarding these RSVP extensions are another important topic of future research.

Bibliography

[AC+93] F. Alvarez-Cuevas et al. Voice synchronization in packet switching networks. In *IEEE Networks Mag 7(5)*, pages 20–25, 1993.

[Alm92] P. Almquist. Type of Service in the Internet Protocol Suite. In *RFC 1349*, July 1992.

[Atk95a] R. Atkinson. IP Authentication Header (AH). In *RFC 1826*, August 1995.

[Atk95b] R. Atkinson. IP Encapsulation Security Payload (ESP). In *RFC 1827*, August 1995.

[Atk95c] R. Atkinson. Security Architecture for the Internet Protocol. In *RFC 1825*, August 1995.

[B+94] R. Braden et al. Integrated Services in the Internet Architecture: an Overview. In *RFC 1633*, June 1994.

[B+97a] M. Borella et al. Analysis of End-to-End Internet Packet Loss: Dependece and Asymmetry. In *Preprint*, 1997.

[B+97b] R. Braden et al. Resource ReSerVation Protocol (RSVP) - Ver 1 Functional Specification. In *RFC 2205*, 1997.

[B+98] S. Blake et al. An Architecture for Differentiated Services. In *IETF Internet Draft (work in progress), draft-ietf-diffserv-arch-02.txt*, October 1998.

[BC95] J.-C. Bolot and H. Crepin. Analysis and Control of Audio Packet Loss over Packet-Switched Networks. In *NOSSDAV*, 1995.

[Ber98] Y. Bernet. A Framework for Use of RSVP with Diff-serv Networks. In *IETF Internet Draft (work in progress), draft-ietf-diffserv-rsvp-01.txt*, November 1998.

[BL+94] T. Berners-Lee et al. Uniform Resource Locators (URL). In *RFC 1738*, December 1994.

[BL+96] T. Berners-Lee et al. Hypertext Transfer Protocol – HTTP/1.0. In *RFC 1945*, May 1996.

[BO97] L. Berger and T. O'Malley. RSVP Extensions for IPSEC Data Flows. In *RFC 2207*, September 1997.

[Bol93] J.-C. Bolot. Characterizing End-to-End Packet Delay and Loss in the Internet. In *High Speed Networks*, volume 2, pages 305–323, 1993.

[BS98] L. Breslau and S. Shenker. Best-Effort versus Reservations: A Simple Comparative Analysis. In *in Proc. of ACM SIGCOMM, Vancover, Canada*, 1998.

[BV98] S. Berson and S. Vincent. Aggregation of Internet Integrated Services State. In *IETF Internet Draft (work in progress), draft-berson-rsvp-aggregation-00.txt*, August 1998.

[BVG97] J.-C. Bolot and A. Vega-Garcia. The Case for FEC-Based Error Control for Packet Audio in the Internet. In *ACM Multimedia Systems*, 1997.

[BVGFPne] J.-C. Bolot, A. Vega-Garcia, and S. Fosse-Parisis. Free Phone, INRIA, http://www.inria.fr/rodeo/fphone/.

[C+89] A. Coleman et al. Subjective Performance Evaluation of the REP-LTP Codec for the Pan-European Cellular Digital Mobile Radio System. In *in Proc. ICASSP*, pages 1075–1079, 1989.

[CCH92] A. Campell, G. Coulson, and D. Hutchison. A Suggested QOS Architecture for Multimedia Communications. Technical report, Dept. of Computing, Lancaster University, U.K., November 1992.

[Co.va] Netscape Communications Co. Netscapes Software Coding Standards Guide for Java, http://developer.netscape.com/tech/java/.

[Co.pt] Netscape Communications Co. Java Script 1.3 Documentation, http://developer.netscape.com/tech/javascript/.

[D+94] B.J. Dempsey et al. On retransmission-based error control for continuous media traffic in packet-switching networks. Technical report, Dept. of Computer Science, University of Virginia, February 1994.

[D+97] M. Degermark et al. Small Forwarding Tables for Fast Routing Lookups. In *in Proc. of ACM SIGCOMM*, 1997.

[D+98a] B. Davie et al. Use of Label Switching with RSVP. In *IETF Internet-Draft (work in progress), draft-ietf-mpls-rsvp-00.txt*, March 1998.

[D+98b] M. Degermark et al. IP Header Compression. In *IETF Internet-Draft (work in progress), draft-degermark-ipv6-hc-06.txt*, June 1998.

[DH95] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. In *RFC 1883, Xerox PARC and Ipsilon Networks*, December 1995.

[DH98] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. In *IETF Internet-Draft (work in progress), draft-ietf-ipngwg-ipv6-spec-v2-02.txt*, August 1998.

[DS78] Y. Dalal and C. Sunshine. Connection Management in Transport Protocols. In *Computer Networks*, volume 2, No. 6, pages 454–473, December 1978.

[Eri93] H. Erikson. MBone: The Multicast Backbone. In *Communications of the ACM*, volume 7, No. 5, pages 8–18, September 1993.

[F+97] R. Fielding et al. Hypertext Transfer Protocol – HTTP/1.1. In *RFC 2068*, January 1997.

[FH98] P. Ferguson and G. Huston. Qualtiy of Service. In *Wiley Computer Publishing*, 1998.

[Fro97] K. Froitzheim. Multimedia-Kommunikation. In *d-punkt Verlag, Heidelberg (Germany)*, 1997.

[FW97] K. Froitzheim and H. Wolf. WebVideo - a Tool for WWW-based Teleoperation. In *International Symposium on Industrial Electronics, Guimaraes, Portugal*, July 1997.

[G.196] ITU G.114. One-Way Transmission Time, ITU-T Recommendation G.114, February 1996.

[GA98] M. Greis and M. Albrecht. Aggregation of Internet Integrated Services State using Paramter-based Admission Control. In *IETF Internet Draft (work in progress), draft-greis-aggregation-with-pbac-00.txt*, November 1998.

[Gadml] F. Gadegast. MPEG-FAQ, http://www.cc.iastate.edu/olc_answers/packages/graphics/mpeg.faq.html.

[Ger87] N. Gerfelder. Graphics Interchange Format (GIF) Specification, CompuServe Incorporated, June 1987.

[GS95] A. Grace and A. Smith. Quality of Service control for adaptive distributed multimedia applications using Esterel. In *Second International Workshop on High Performance Protocol Architectures, Sydney*, December 1995.

[H+95] V. Hardman et al. Reliable audio for use over the Internet. In *in Proc. of INET'95, Honolulu, Hawaii*, June 1995.

[H+98] M. Handley et al. SIP: Session Iinitiation Protocol. In *IETF Internet Draft (work in progress), draft-ietf-mmusic-sip-10.txt*, May 1998.

[H.293] ITU H.261. Video Codec for Audiovisual Services at px64 kbps, ITU-T Recommendation H.261, March 1993.

[H.396] ITU H.323. Visual telephone systems and equipment for local area networks which provide a non-guaranteed quality of service, ITU-T Recommendation H.323, May 1996.

[Han98] M. Handley. SDP: Session Description Protocol. In *RFC 2327, ISI/LBNL*, April 1998.

[Hui97] C. Huitema. IPv6 - The new Internet Protocol. In *2nd edition, Prentice Hall*, 1997.

[ISI98] ISI. RSVP Daemon Release 2a3, ftp://ftp.isi.edu/, August 1998.

[Jay80] N. S. Jayant. Effects of packet loss on waveform coded speech. In *in Proc. of the 5th Data Communications Symposium*, pages 275–280, Atlant, Ga. 1980.

[JMat] V. Jacobson and S. McCanne. VAT - LBNL Audio Conferencing Tool, http://www-nrg.ee.lbl.gov/vat/.

[JTC93] ISO IEC JTC1. Information Technology - Digital Compression and Coding of Continuous-Tone still Images, International Standard ISO/IEC IS 10918, 1993.

[K+98] D. Katz et al. IPv6 Router Alert Option. In *IETF Internet Draft (work in progress), draft-ietf-ipngwg-ipv6router-alert-04.txt*, February 1998.

[Kat97] D. Katz. IP router alert option. In *RFC 2113,*, February 1997.

[M+98] J. Martens et al. Voice over IP: the impact of RSVP. In *in Proc. of SPIE*, volume 3529, Internet Routing and Quality of Service, November 1998.

[MD90] J. Mogul and S. Deering. Path MTU Discovery. In *RFC 1191*, November 1990.

[MHSC99] L. Mathy, D. Hutchison, S. Schmid, and G. Coulson. Improving RSVP for Better Support of Internet Multimedia Communications. In *presented at ICMCS'99, Florence, Italy*, 1999.

[MJ95] S. McCanne and V. Jacobson. vic: A flexible framework for packet video. In *in Proc. of ACM Multimedia '95*, November 1995.

[MKT98] S.B. Moon, J. Kurose, and D. Towsley. Packet audio playout delay adjustment: performance bounds and algorithms. In *ACM Multimedia Systems (1998)6*, pages 17–28, 1998.

[Mon83] W.A. Montgomery. Techniques for packet voice synchronization. In *IEEE Select Areas Communication 6(1)*, pages 1022–1028, 1983.

[MSH98] L. Mathy, S. Schmid, and D. Hutchison. REDuced Overhead RSVP. Technical report, Dept. of Computing, Lancaster University, U.K., September 1998.

[N+97] K. Nichols et al. A Two-bit Differentiated Services Architecture of the Internet. In *IETF Internet Draft (work in progress), draft-nichols-diff-svc-arch-00.txt*, November 1997.

[Pax96a] V. Paxson. End-to-End Routing Behavior in the Internet. In *in Proc. IEEE/ACM Transactions on Networking 5(5)*, pages 601–615, 1996.

[Pax96b] V. Paxson. End-to-End Routing Behavior in the Internet. In *in Proc. IEEE/ACM Transactions on Networking, Vol. 5, no. 5*, pages 601–615, 1996.

[PG93] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The multiple node case. In *in Proc. of the 12th Annual Joint Converence of the IEEE Computer and Communications Societies on Networking*, volume 2, pages 521–530, March 1993.

[Pos80a] J. Postel. User Datagram Protocol. In *RFC 768, ISI*, August 1980.

[Pos80b] J. Postel. DOD Standard Internet Protocol. In *RFC 760, DARPA*, January 1980.

[Pos80c] J. Postel. Transmission Control Protocol. In *RFC 761, DARPA*, January 1980.

[Pos81] J. Postel. Internet Protocol. In *RFC 791, DARPA*, September 1981.

[PS98] P. P. Pan and H. Schulzrinne. YESSIR: A simple reservation mechanism for the Internet. In *in Proc. International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, July 1998.

[R+94] R. Ramjee et al. Adaptive playout mechanisms for packetized audio applications in wide-are networks. In *in Proc. of IEEE Infocom '94, Montreal, Canada*, 1994.

[R+98a] E.C. Rosen et al. Multiprotocol Label Switching Architecture. In *IETF Internet-Draft (work in progress), draft-ietf-mpls-arch-02.txt*, July 1998.

[R+98b] J. Rosenberg et al. Elevating RTP to Protocol Status. In *IETF Internet Draft (work in progress), draft-rosenberg-rtpproto-00.txt*, September 1998.

[Riz97] L. Rizzo. The FreeBSD Audio Driver. Technical report, University of Pisa, Italy, 1997.

[RP94] J. Reynolds and J. Postel. Assigned Numbers. In *RFC 1700*, October 1994.

[S+96] H. Schulzrinne et al. RTP: A Transport Protocol for Real-Time Applications. In *RFC 1889*, January 1996.

[S+97] D. Sanghi et al. Experimental assessment of end-to-end behavior on the Internet. In *in Proc. of IEEE Infocom '93, San Francisco, CA*, pages 867–887, 1997.

[S+98a] S. Schmid et al. QoS-based real-time audio streaming in IPv6 networks. In *in Proc. of SPIE*, volume 3529, Internet Routing and Quality of Service, November 1998.

[S+98b] H. Schulzrinne et al. Real Time Streaming Protocol (RTSP). In *RFC 2326*, April 1998.

[Sch92] H. Schulzrinne. Voice communication across the Internet: A network voice terminal. Technical report, Dept. of Computer Science, University of Massachusetts, July 1992.

[Sch97] U. Schwantag. An Analysis of the Applicability of RSVP. Technical report, Institute of Telematics, University of Karlsruhe (Germany), July 1997.

[Sch98] H. Schulzrinne. RTP Profile for Audio and Video Converences with Minimal Control. In *IETF Internet-Draft (work in progress), draft-ietf-avt-profile-new-04.txt*, November 1998.

[SDRS98] S. Schmid, M. Dunmore, N. Race, and A. Scott. RSVP Extensions for Flow Labels. In *IETF Internet-Draft (work in progress), draft-schmid-rsvp-fl-01.txt*, August 1998.

[SM90] N. Shacham and P. McKenney. Packet recovery in high-speed networks using coding and buffer management. In *in Proc. IEEE Infocom' 90, San Fransisco, CA*, pages 124–131, May 1990.

[SPG97] S. Shenker, C. Partridge, and R. Guerin. Specification of Guaranteed Quality of Service. In *RFC 2212*, September 1997.

[Ste97] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. In *RFC 2001, NOAO*, January 1997.

[T+96] Andreas Tirakis et al. Distributed Multimedia Architectures - State-of-the-Art Report. Technical report, Project: Distributed Video Production (DVP), September 1996.

[Tan96] A. Tanenbaum. Computer Networks. In *3rd edition, Prentice Hall*, 1996.

[Wro97a] J. Wroclawski. Specification of the Controlled-Load Network Element Service. In *RFC 2211, MIT LCS*, September 1997.

[Wro97b] J. Wroclawski. The Use of RSVP with IETF Integrated Services. In *RFC 2210, MIT LCS*, September 1997.

[Z+93] L. Zhang et al. RSVP: A new Resource ReSerVation Protocol. In *IEEE Networks*, volume 7, No. 5,, September 1993.

[ZK98] D. Zappala and J. Kann. RSRR – A Routing Interface for RSVP. In *IETF Internet-Draft (work in progress), draft-ietf-rsvp-routing-02.txt*, July 1998.

Name: Stefan Schmid Matr. Nr. 280347

Erklärung

Ich erkläre, daß ich die Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ....
(Unterschrift)