TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering

Harri Marjamäki

DELAY CHARACTERISTICS OF AN IP VOICE TERMINAL

Master's thesis submitted for official examination for the degree of Master of Science in Engineering.

Tampere, 13 January, 1999

Examiner: Professor Markku Renfors

FOREWORD

This thesis has been done at the telecommunications laboratory of Helsinki University of Technology as a part of IPANA (IP/ATM Network Architectures) research project in the period of May-December 1998.

I would like to thank professor Raimo Kantola from Helsinki University of Technology and Osmo Eerola from Miratel Oy who provided me the opportunity to work on this topic. I also thank professor Markku Renfors from Tampere University of Technology and professor Raimo Kantola from Helsinki University of Technology for their constructive comments on the thesis.

I want to thank Markus Peuhkuri for all his help with the measurements and also the rest of the personnel in the telecommunications laboratory for their co-operation, especially Veikko, Jose and Vesa. Finally, I thank my girlfriend Tarja for her mental support.

Tampere, 13 January, 1999

Harri Marjamäki

Toripolku 1

21500 PIIKKIÖ

puh. 040-5839797

TABLE OF CONTENTS

TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering

Telecommunications Laboratory

MARJAMÄKI, HARRI: Delay Characteristics of an IP Voice Terminal

Master of Science Thesis, 93 pages, 41 enclosure pages

Examiner: Professor Markku Renfors

Financed by: TEKES, Miratel Oy

January 1999

Keywords: Voice over IP, Speech coding, Playout algorithms, Unix

This thesis examines issues related to the transmission of voice over packet networks using the Internet Protocol (IP). We focus on studying the delays that are generated in the terminal, which is a Unix workstation equipped with an IP voice application software.

Delay components in the terminal are presented. We measure the processing delays in the terminal using different audio codecs and measure the end-to-end delays using different scheduling parameters for the IP voice application. A significant part of the delay is shown to be caused by buffering at the receiving terminal. This is a feature of the application software and can be removed by modifying the source code.

We also make a comparison of adaptive algorithms that are used to calculate the playout times of the voice packets. Algorithms are simulated using different network loads and thus different delay distributions of the voice packets in an Ethernet. We present a new playout algorithm which is a combination of two existing algorithms. This algorithm is shown to outperform the other two real-time algorithms that are compared in our studies.

TAMPEREEN TEKNILLINEN KORKEAKOULU

Sähkötekniikan osasto

Tietoliikennetekniikka

MARJAMÄKI, HARRI: Viiveiden tarkastelu IP-päätelaitteessa

Diplomityö, 93 s., 41 liitesivua

Työn valvoja: Professori Markku Renfors

Rahoittajat: TEKES, Miratel Oy

Tammikuu 1999

Avainsanat: Voice over IP, Speech coding, Playout algorithms, Unix

Tässä työssä tutkittavat asiat liittyvät puheen välittämiseen pakettikytkentäisten verkkojen yli käyttäen Internet protokollaa (IP). Perusongelmana IP-protokollan käytössä puheen siirtoon ovat reaaliaikaisten sovellusten vaatimukset alhaisesta viiveestä ja viiveen vaihtelusta. Ruuhkaisessa Internetissä lisäongelmia tuo vielä vaadittu suuri kaista. Puheen koodaaminen ja pakkaaminen tuo helpostusta sekä viive- että kaistavaatimuksiin, mutta vaatii paljon prosessointia.

Aikaisemmissa tutkimuksissa on todettu merkittävän osan viiveestä syntyvän päätelaitteessa. Päätelaitteena toimii useimmiten PC tai työasema, joka on varustettu jollain IP-puheohjelmistolla. Lisäksi tarvitaan äänikortti, mikrofoni ja kaiutin sekä tietysti verkkoyhteys Internetiin.

Päätelaitteessa viivettä syntyy sekä laitteiston, että myös ohjelmiston toiminnasta. Laitteistossa puhenäytteet muunnetaan analogisesta digitaaliseen muotoon äänikortilla ennen lähetystä. Vastaavasti vastaanottavassa päätelaitteessa suoritetaan muunnos digitaalisesta analogiseen muotoon. IP-puheohjelmiston asetuksilla määritellään puhepakettien pituus sekä käytetty puheenkoodausalgoritmi. Puheen koodauksesta ja dekoodauksesta aiheutuvat prosessointiviiveet riippuvat käytetystä algoritmista sekä päätelaitteen prosessorin laskentatehosta. Myös käyttöjärjestelmällä ja sen käyttämällä skedulointimenetelmällä on vaikutusta viiveeseen, koska käyttöjärjestelmä joutuu jakamaan prosessoriaikaa päätelaitteessa kullakin hetkellä käynnissä olevien ohjelmien kesken.

Verkon aiheuttama viiveen vaihtelu joudutaan vastaanottavassa päätelaitteessa poistamaan puskuroimalla paketteja muistiin. Koska yleensä puheen hiljaisia jaksoja ei lähetetä, lähetys koostuu jaksoista puhetta. Puskurointiviiveet lasketaan uudelleen kullekin jaksolle käyttäen adaptiivista algoritmia. Käytetyllä algoritmilla ja sen parametreilla on merkittävä vaikutus kokonaisviiveeseen.

Tässä työssä suoritetaan viivemittauksia Unix-käyttöjärjestelmällä varustetussa työasemassa, joka yhdessä Nevot-ohjelmiston kanssa toimii IP-päätelaitteena. Nevotin lähdekoodit instrumentoidaan siten, että tuloksena saatavista profilointitiedostoista nähdään ohjelman kuluttama prosessoriaika erilaisilla audiokoodauksilla. Kokonaisviiveet kahden työaseman välillä mitataan oskilloskoopilla ja viiveiden riippuvuus ohjelmalle käytetyistä skedulointiparametreista todetaan. Myös muut viiveen komponentit, kuten verkon viive ja viiveet äänikortilla mitataan.

Merkittävä osa kokonaisviiveestä huomataan syntyvän vastaanottavassa työasemassa, jossa puskurointiviive kasvaa pian ohjelman käynnistämisen jälkeen 60 millisekuntiin, vaikka verkko olisi kuormittamaton. Tämä johtuu Nevotin ominaisuudesta, jossa tarkoituksena on ennaltaehkäistä puskurin alivuotoja ja täten parantaa äänen laatua. Tämä ominaisuus on helppo poistaa muuttamalla lähdekoodia ja kääntämällä ohjelma uudelleen. Verkon ollessa kuormittamaton tämä muutos ei vaikuta äänen laatuun ja kokonaisviive saadaan pudotettua 30-40 millisekunnin tasolle, jos käytetään 20 millisekunnin pakettikokoa.

Työn loppuosassa vertaillaan puskurointiviiveiden laskemiseen käytettäviä adaptiivisia algoritmeja. Kuormittamalla verkkoa protokolla-analysaattorilla saadaan aiheutettua erilaisia pakettien viivejakautumia. Lähetettyjen ja vastaanotettujen pakettien RTP-aikaleimat talletetaan tiedostoiksi käyttäen Nevotin debug-optiota, jolloin eri algoritmeja voidaan simuloida Matlabin avulla käyttäen samoja datatiedostoja.

Työssä esitetään uusi algoritmi, joka on toteutettu yhdistämällä kaksi lähdekirjallisuudesta löydettyä algoritmia. Tämän algoritmin osoitetaan antavan parempia tuloksia verrattuna kahteen muuhun reaaliaikaiseen algoritmiin, joita on vertailtu simuloinneissa.

TABLE OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

	A
ACELP	Algebraic Code Excited Linear Prediction
ACR	Absolute Category Rating
ADPCM	Adaptive Differential Pulse Code Modulation
	C
CCR	Comparison Category Rating
CELP	Code Excited Linear Prediction
CIF	Common Intermediate Format
CMOS	Comparison Mean Opinion Score
CPU	Central Processor Unit
CS-ACELP	Conjugate Structure - Algebraic Code Excited Linear Prediction
CSRC	Contributing Source
	D
DC	Direct Current
DSI	Digital Speech Interpolator
DSP	Digital Signal Processor
	G
GSM	Global System for Mobile communications
	H
HW	Hardware
	I
IHL	Internet Header Length
IP	Internet Protocol
IPX	Internet Packet Exchange Protocol
ISDN	Integrated Services Digital Network
ISN	Initial Sequence Number
ISO	International Standards Organization
ITU-T	International Telecommunications Union – Telecom Sector
	L
LAN	Local Area Network
LD-CELP	Low Delay - Code Excited Linear Prediction
LLC	Logical Link Control
LP	Linear Prediction
LPC	Linear Predictive Coding
LSP	Linear Spectrum Pair
	M
MA	Moving Average
MCU	Multipoint Control Unit
MIPS	Million Instructions Per Second
MMX	MultiMedia eXtension
MOS	Mean Opinion Score
MP-MLQ	MultiPulse - Maximum Likelihood Quantization
MTU	Maximum Transmission Unit
	P
PBX	Private Branch Exchange

PC	Personal Computer
PCM	Pulse Code Modulation
PSI-CELP	Pitch Synchronous Innovation - Code Excited Linear Prediction
PSTN	Public Switched Telephone Network
PSVQ	Predictive Split Vector Quantizer
	Q
QCELP	Qualcomm Code Excited Linear Prediction
QCIF	Quarter Common Intermediate Format
	R
RAM	Random Access Memory
ROM	Read Only Memory
RPE-LTP	Regular Pulse Excitation - Long Term Prediction
RT	Real Time
RTCP	Real Time Control Protocol
RTP	Real Time Protocol
	S
SNAP	SubNetwork Access Protocol
SPX	Sequenced Packet Exchange Protocol
SSRC	Synchronization Source
SYS	System
	T
TASI	Time Assigned Speech Interpolator
TCP	Transmission Control Protocol
TDM	Time Division Multiplexing
TPDU	Transmission Protocol Data Unit
TS	Time Sharing
	U
UDP	User Datagram Protocol
	V
VCELP	Vector Code Excited Linear Prediction
VoIP	Voice over IP
VQ	Vector Quantization

1. INTRODUCTION

The transmission of voice over packet switched networks was an active research area in the late 70’s and the early 80’s. Much of the work then focused on using packet switching for both voice and data in a single network. Packet voice, and more generally, packet audio applications have recently become again of interest. This interest has been fueled by the availability of supporting hardware and increased bandwith throughout the Internet.

The Internet provides a simple single class best effort service. From a connection’s point of view, the best effort service amounts in practice to offering a channel with time-varying characteristics such as delay and loss distributions. These characteristics are not known in advance since they depend on the behavior of other connections throughout the network. A variety of audio tools have been available for a few years, and they have been used to audiocast conferences. Experimental evidence suggests that, although the quality of the audio delivered by Internet tools has improved, audio quality is still mediocre in many audio conferences. This is clearly a concern since audio quality has been found to be more important than video quality or audio/video synchronization to succesfully carry out collaborative work.

For audio quality in packet audio applications, the main concerns are the delay and delay variance. In earlier studies [1] it was noted that in LANs and campus networks where network caused delay and delay variance were relatively small, most of the end-to-end delay was accumulated in the terminals.

In the terminal, delay is accumulated both by the hardware and the software. In the audio hardware, voice samples are A/D converted at the sender and D/A converted at the receiver. In the packet audio application software processing delay is introduced. Processing delay is very much dependent on the used speech codec. Some codecs, like PCM codec have very little to do and introduce very little delay whereas, for example, a GSM codec requires excessive computation and causes significantly more processing delay. Buffering of voice samples is necessary both at sending and receiving end. Buffering delays are introduced both in the audio hardware and in the packet audio software. Delays are introduced also by the operating system because it has to assign processor time also to other processes that are simultaneously running in the terminal.

The network caused delay variance has to be smoothed in the application software in order to preserve the sound quality. Voice packets are buffered at the receiver and they are played out periodically. The algorithms used to calculate the appropriate playout time for each packet of voice are called playout algorithms.

1.1. Goals of the thesis

The purpose of this thesis is to study the delays that are generated in an IP voice terminal during a packet audio connection. The objectives are to find out what are the delay components in the terminal, study a VoIP application software in detail to resolve if there is something in the implementation that causes additional delays, and study how well Unix operating system supports real-time applications. We will also make a literature study of playout algorithms and simulate them with Matlab.

1.2. Structure of the thesis

Chapter two begins with an introduction to the main concepts related to Voice over IP: its applications, the protocols and the standards. In the same chapter we will present some of the fundamentals of packet voice: voice traffic models and the voice packetization process. Chapter three is about speech coding. It begins with introduction to speech coder attributes and speech coding techniques. Then the most used speech codecs in packet voice applications are presented. Chapter four explains playout delay adaptation: the mathematics and algorithms used in delay estimation. Chapter five is about operating systems and scheduling concepts, the Unix process scheduler is explained in more detail. Chapter six reports the performed measurements and obtained results. Chapter seven presents conclusions and topics for future work.

2. VOICE OVER IP CONCEPTS

Voice over IP (VoIP) is the transmission of voice over networks using the Internet Protocol. IP-networks have become increasingly popular in the past few years, the exponential growth of the public Internet leading the way into the IP-world.

In this chapter, the basic concepts relating to voice over IP will be presented: history of Voice over IP, what it is used for and why, the applications, the main standards and protocols relating to VoIP: IP, UDP, TCP, RTP, and H.323.

2.1. History of Voice over IP

The transmission of voice signals over packet networks is not a new concept. It was an active research topic in the late 70's and early 80's. Already then, in the late 70's there was discussion and even experiments with packetized voice over the ARPANET, the predecessor of the Internet using IP and specialized coding and packetizing equipment [2]. The conclusion then was that packetized voice has economical advantages and can be done. Still it took some fifteen years for the packetized voice to gain popularity. Specialized equipment are no longer required: a desktop general purpose personal computer, a sound card, microphone and speakers, and some software are all that is needed. The needed equipment is already built into most multimedia capable computers today, and with the wide-spread connectivity to the Internet, local and wide area telephony over packet data networks is possible for a very large audience.

2.2. Benefits of Voice over IP

Increased bandwith and computational resources have made interactive voice communication between workstations across packet communication facilites feasible. Cooperative work, teleconferencing, and videotelephones are applications that have attracted a large amount of implementation and research interest.

Transmitting voice across a packet-switched network offers a number of advantages over the circuit-switched approach. First, we obtain all the well-known benefits of service integration, particularly important in a multimedia setting. Secondly, we may be able to achieve a higher bandwith utilization since voice does not always use its peak bandwith (due to silence periods and variable rate coding). Finally, because interleaving several associations tends to be easier in a packet-switched network, control can be more sophisticated [3].

2.3. Scenarios of Voice over IP

There are three basic network scenarios for Voice over IP. These scenarios are illustrated in Figure 2-1.

· Computer to Computer

This is the basic scenario: both A and B subscribers are using computers attached to an IP network as terminals.

· Computer to Phone and Phone to Computer

In this scenario, one of the subscribers is using a computer for IP-voice and the other uses a phone on a PSTN/ISDN/GSM/TDM network. A gateway on the edge of the IP network translates IP-voice to voice and takes care of the signaling between the two networks.

· Phone to Phone

Both subscribers are using conventional phones in this option, and the IP network is used for the long distance connection. Gateways on both ends take care of translations between networks [1].

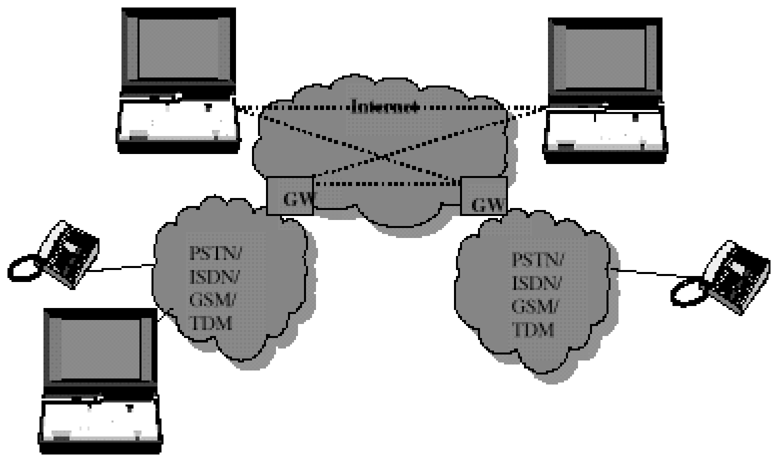


Figure 2-1: The framework for IP voice

2.4. Voice over IP terminal equipment

The IP phone terminal equipment can be workstations or personal computers equipped with IP phone software, sound card, speakers, a microphone and of course a network interface. The first experimental terminal software was designed for UNIX workstations with plenty of processing capability and memory, but the Pentium equipped PCs of today can do the job as well. The minimum requirements are mostly stated as the 486. If video is used with audio, then Pentium II MMX processor equipped computer could be barely sufficient.

Nowadays there are also available special IP-telephones that look like normal telephones with a telephone handset. An ordinary telephone handset gives more comfort and better audio quality than the speaker/microphone combination.

2.5. IP Voice gateway

An IP voice gateway is an interworking unit capable of translating IP-phone signals to ordinary telephone signals, capable of IP address to telephone number conversion and foremost can signal open a connection from the IP network to a terminal in the telephone network. So far the gateways have been built around PC hardware. The processing intensive coding and decoding of voice is done with special DSP-equipped telephony boards. The control functions are run in main memory using the CPU. There are scaling and reliability problems in architectures like this, and the cost is also relatively high [1].

2.6. IP Voice related standards and protocols

In this section we present the most important protocols and standards related to Voice over IP. Figure 2-2 presents the Voice over IP protocol stack. We will start with Internet Protocol. This will be continued by an introduction to UDP and TCP protocols, the two transport layer protocols of IP. For realtime transmission of voice UDP is mostly used. TCP is used in relation to VoIP for streaming control and applications where added buffering delay is acceptable, e.g. audio broadcast. To make use of TCP retransmissions, buffering is needed. Waiting for retransmissions is not appropriate for realtime communications, where interactivity needs to be maintained. RTP is the session layer protocol used for synchronization, multicast session participant information relaying, and recipient network quality monitoring purposes.

RTP
TCP/UDP
IP
LLC/SNAP
Physical layer

Figure 2-2: The Voice over IP protocol stack

2.6.1. The Internet Protocol (IPv4)

2.6.1.1. IP Services

Two primitives are defined at the user-IP interface. The Send primitive is used to request transmission of a data unit. The Deliver primitive is used by IP to notify a user of the arrival of a data unit. Although not part of the standard, IP is expected to use an Error primitive to notify a user of failure in providing the requested service. This service is not assumed to be reliable, that is, there is no guarantee that errors will be reported.

2.6.1.2. IP Protocol

The protocol between IP entities is best described by defining the IP datagram format, which is shown in Figure 2-3. The fields are:

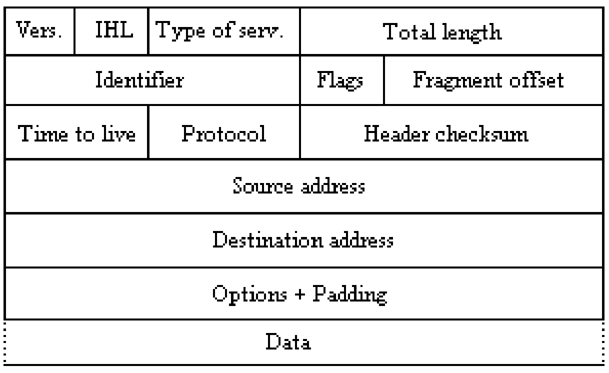


Figure 2-3: IP protocol data unit

- Version (4 bits): version number, included to allow evolution of the protocol.
- Internet header length (IHL) (4 bits): length of header in 32-bit words. The minimum value is 5. Thus a header is at least 20 octets long.
- Type of service (8 bits): specifies reliability, precedence, delay, and troughput parameters.
- Total length (16 bits): total data unit length, including header, in octets.
- Identifier (16 bits): together with source address, destination address, and user protocol, intended to uniquely identify a datagram.
- Flags (3 bits): one bit, the More flag, used for fragmentation (segmentation) and reassembly. Another bit, if set, prohibits fragmentation. This facility may be useful if it is known that the destination does not have the capability to reassemble fragments. The third bit is currently not used.
- Fragment offset (13 bits): indicates where in the datagram this fragment belongs. It is measured in 64 bit units. This implies that fragments (other than the last fragment) must contain a data field that is a multiple of 64 bits long.
- Time to live (8 bits): measured in router hops.
- Protocol (8 bits): indicates the next level protocol which is to receive the data field at the destination.
- Header checksum (16 bits): frame check sequence on the header only. Since some header fields may change, this is reverified and recomputed at each router.
- Source address (32 bits): coded to allow a variable allocation of bits to specify the network and the station within the specified network.
- Destination adress (32 bits): as above.
- Options (variable): encodes the options requested by the sender.
- Padding (variable): used to ensure that the header ends on a 32-bit boundary
- Data (variable): the data field must be a multiple of eight bits in legth. Total length of data field plus header is a maximum of 65535 octets.

It should be easy to see how the services specified above map into the fields of the IP data units [4].

2.6.2. The next generation of IP (IPv6)

IP next generation (IPng) [5], [6] is a new version of the Internet Protocol designed as a successor to the IP version 4. The version number assigned for IPng is 6 and it is formally called IPv6. It was not designed to be a giant leap away from IPv4 - many of the functions of IPv4 were kept. The primary motivation for the design of IPng was the exponential growth of the Internet leading to running out of IP address space. IPng offers: expanded routing and addressing capabilities, auto-configuration of addresses, improved scalability of multicast routing by adding a "scope" field to multicast addresses, a simplified header format (despite the increased address size, headers have only doubled), improved support for options, quality-of-service capabilities (flow labeling), authentication and privacy capabilities. The IPng header (Fig. 2-4) consists of two parts, the basic IPng header and IPng extension headers.

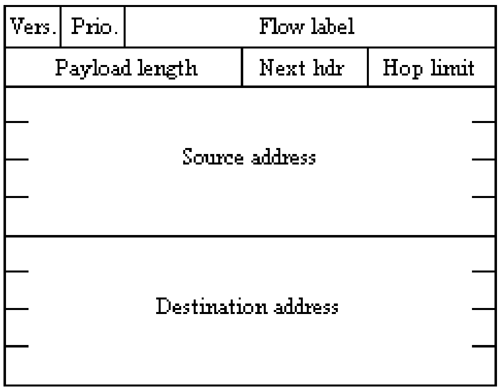


Figure 2-4: IPv6 header

2.6.3. Transport Control Protocol (TCP)

2.6.3.1. TCP Services

TCP is designed to provide reliable communication between pairs of processes (TCP users) across a variety of reliable and unreliable networks and Internets. Functionally, it is equivalent to class 4 ISO Transport. In contrast to the ISO model, TCP is stream oriented. That is, TCP users exchange streams of data. The data are placed in allocated buffers and transmitted by TCP in segments (TPDUs). TCP supports security and precedence labeling. In addition, TCP provides two useful facilities for labeling data: push and urgent:

- Data stream push: Ordinarily, TCP decides when sufficient data has accumulated to form a TPDU for transmission. The TCP user can require TCP to transmit all outstanding data up to and including that labeled with a push flag. On the receiving end, TCP will deliver these data to the user in the same manner. A user might request this if it has come to a logical break in the data.

- Urgent data signaling: This provides a means of informing the destination TCP user that significant or “urgent” data is in the upcoming data stream. It is up to the destination user to determinate appropriate action.

When user wants some service from TCP, it sends a service request to TCP, and TCP will reply with a service response.

2.6.3.2. TCP header format

TCP uses only a single type of TPDU, called a TCP segment. The header is shown in Figure 2-5. Because one header must serve to perform all protocol machanisms, it is rather large. The TCP header is a minimum of 20 octets long. The fields are [4]:

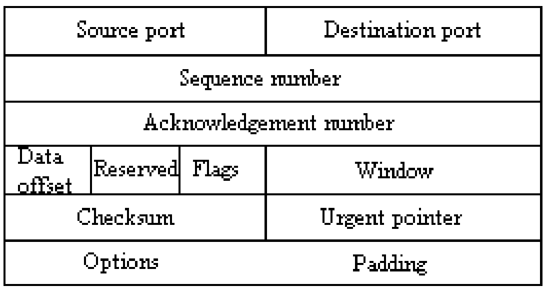


Figure 2-5: TCP header

- Source port (16 bits): identifies source service access point.
- Destination port (16 bits): identifies destination service access point
- Sequence number (32 bits): sequence number of the first data octet in this segment except when SYN is present. If SYN is present, it is the initial sequence number (ISN) and the first data octet is ISN+1.
- Acknowledgement number (32 bits): a piggybacked acknowledgement. Contains the sequence number of the next octet that the TCP entity expects to receive.
- Data offset (4 bits): number of 32-bit words in the header.
- Reserved (6 bits): reserved for future use.
- Flags (6 bits):

  URG: urgent pointer field significant

  ACK: acknowledgement field significant

  PSH: push function

  SYN: synchronize the sequence numbers

  FIN: no more data from the sender

- Window (16 bits): flow control credit allocation, in octets. Contains the number of data octets beginning with the one indicated in the acknowledgment field which the sender is willing to accept.
- Checksum (16 bits): the one’s complement of the sum modulo 2^{16}-1 of all the 16-bit words in the segment plus a pseudo-header.
- Urgent pointer (16 bits): points to the octet following the urgent data. This allows the receiver to know how much urgent data is coming.
- Options (variable): at present, only one option is defined, which specifies the maximum TPDU size that will be accepted.

TCP is designed specifically to work with IP. Hence, some user parameters are passed down by TCP to IP for inclusion in the IP header. This TCP/IP linkage means that the required minimum overhead for every data unit is actually 40 octets.

2.6.4. User datagram protocol (UDP)

UDP provides a connectionless service for application-level procedures. Thus UDP is basically an unreliable service; delivery and duplicate protection are not guaranteed. However, this does reduce the overhead of the protocol and may be adequate in many cases. An example of the use of UDP is in the context of network management.

UDP sits on top of IP. Because it is connectionless, UDP has very little to do. Essentially, it adds a port addressing capability to IP. This is best seen by examining the UDP header, shown in Figure 2-6. The header includes a source port and destination port. The length field contains the length of the entire UDP segment, including header and data. The checksum is the same algorithm used for TCP and IP. For UDP, the checksum applies to the entire UDP segment plus a pseudo-header prefixed to the UDP header at the time of calculation and is the same one used for TCP. If an error is detected, the segment is discarded and no further action is taken.

The checksum field in UDP is optional. If it is not used, it is set to zero. However, it should be pointed out that the IP checksum applies only to the IP header and not to the data field, which in this case consists of the UDP header and the user data. Thus, if no checksum calculation is performed by UDP, then no check is made on the user data [4].

Source port	Destination port
Length	Checksum

Figure 2-6: UDP header

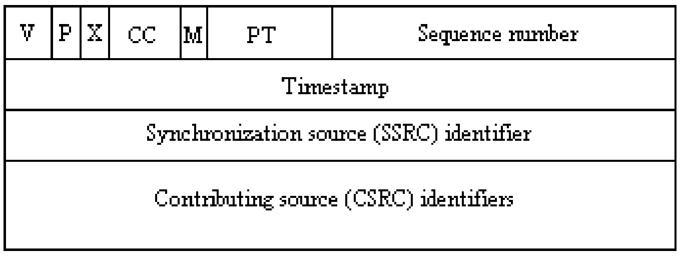
2.6.5. Realtime protocol (RTP)

RTP provides end-to-end delivery services for data with real-time characteristics, such as interactive audio and video. Those services include payload type identification, sequence numbering, timestamping, and delivery monitoring. Applications typically run RTP on top of UDP to make use of its multiplexing and checksum services; both protocols contribute parts of the transport protocol functionality. However, RTP may be used with other suitable underlying network or transport protocols. RTP supports data transfer to multiple destinations using multicast distribution if provided by the underlying network.

Note that RTP itself does not provide any mechanism to ensure timely delivery or provide other quality-of-service guarantees, but relies on lower-layer services to do so. It does not guarantee delivery or prevent out-of-order delivery, nor does it assume that the underlying network is reliable and delivers packets in sequence. The sequence numbers included in RTP allow the receiver to reconstruct the sender's packet sequence, but sequence numbers might also be used to determine the proper location of a packet, for example in video decoding, without necessarily decoding packets in sequence [7].

2.6.5.1. RTP Data Transfer Protocol

The RTP header has the following format (Fig. 2-7):



V	P	X	CC	M	PT	Sequence number
Timestamp						
Synchronization source (SSRC) identifier						
Contributing source (CSRC) identifiers						

Figure 2-7: RTP header

The first twelve octets are present in every RTP packet, while the list of CSRC identifiers is present only when inserted by a mixer. The fields have the following meaning:

- version (V): 2 bits. This field identifies the version of RTP. The version defined by this specification is two.
- padding (P): 1 bit. If the padding bit is set, the packet contains one or more additional padding octets at the end which are not part of the payload. Padding may be needed by some encryption algorithms with fixed block sizes or for carrying several RTP packets in a lower-layer protocol data unit.
- extension (X): 1 bit. If the extension bit is set, the fixed header is followed by exactly one header extension.
- CSRC count (CC): 4 bits. The CSRC count contains the number of CSRC identifiers that follow the fixed header.
- marker (M): 1 bit. The interpretation of the marker is defined by a profile. It is intended to allow significant events such as frame boundaries to be marked in the packet stream.
- payload type (PT): 7 bits. This field identifies the format of the RTP payload and determines its interpretation by the application.
- sequence number: 16 bits. The sequence number increments by one for each RTP data packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence.
- timestamp: 32 bits. The timestamp reflects the sampling instant of the first octet in the RTP data packet. The sampling instant must be derived from a clock that increments monotonically and linearly in time to allow synchronization and jitter calculations. The resolution of the clock must be sufficient for the desired synchronization accuracy and for measuring packet arrival jitter.
- SSRC: 32 bits. The SSRC field identifies the synchronization source. This identifier is chosen randomly, with the intent that no two synchronization sources within the same RTP session will have the same SSRC identifier. Although the probability of multiple sources choosing the same identifier is low, all RTP implementations must be prepared to detect and resolve collisions.
- CSRC list: 0 to 15 items, 32 bits each. The CSRC list identifies the contributing sources for the payload contained in this packet. The number of identifiers is given by the CC field. If there are more than 15 contributing sources, only 15 may be identified. CSRC identifiers are inserted by mixers, using the SSRC identifiers of contributing sources. For example, for audio packets the SSRC identifiers of all sources that were mixed together to create a packet are listed, allowing correct talker indication at the receiver.

SSRC and CSRC are of course not relevant in a unicast session, if the source and the receiver can be identified using network protocol source field. For example, in a two person call in the Internet, the source is identified by IP source address.

2.6.5.2. RTP Control Protocol (RTCP)

The RTP control protocol (RTCP) is based on the periodic transmission of control packets to all participants in the session, using the same distribution mechanism as the data packets. The underlying protocol must provide multiplexing of the data and control packets, for example using separate port numbers with UDP. RTCP performs four functions:

1. The primary function is to provide feedback on the quality of the data distribution. This is an integral part of the RTP's role as a transport protocol and is related to the flow and congestion control functions of other transport protocols. The feedback may be directly useful for control of adaptive encodings, but experiments with IP multicasting have shown that it is also critical to get feedback from the receivers to diagnose faults in the distribution. This feedback function is performed by the RTCP sender (Figure 2-8) and receiver (Figure 2-9) reports.

2. RTCP carries a persistent transport-level identifier for an RTP source called the canonical name or CNAME. Since the SSRC identifier may change if a conflict is discovered or a program is restarted, receivers require the CNAME to keep track of each participant. Receivers also require the CNAME to associate multiple data streams from a given participant in a set of related RTP sessions, for example to synchronize audio and video.

3. The first two functions require that all participants send RTCP packets, therefore the rate must be controlled in order for RTP to scale up to a large number of participants. By having each participant send its control packets to all the others, each can independently observe the number of participants. This number is used to calculate the rate at which the packets are sent.

4. A fourth, optional function is to convey minimal session control information, for example participant identification to be displayed in the user interface.

Functions 1-3 are mandatory when RTP is used in an IP multicast environment, and are recommended in all environments.

Several RTCP packet types are defined to carry a variety of control information:

- SR: Sender report, for transmission and reception statistics from participants that are active senders
- RR: Receiver report, for reception statistics from participants that are not active senders
- SDES: Source description items, including CNAME
- BYE: Indicates end of participation
- APP: Application specific functions

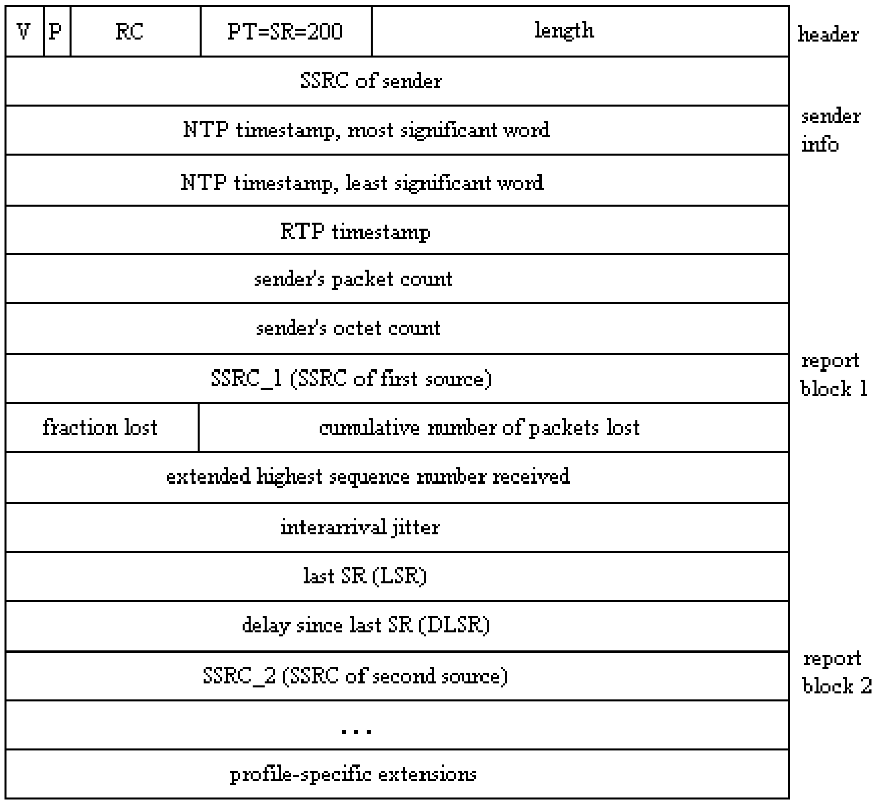
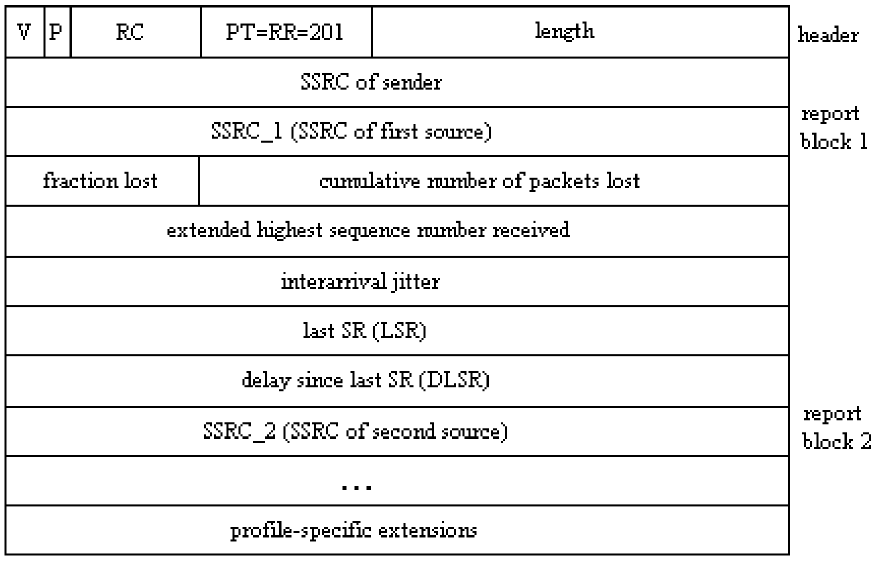

V
P
RC
PT=SR=200
length
header
SSRC of sender
NTP timestamp, most significant word
sender info
NTP timestamp, least significant word
RTP timestamp
sender's packet count
sender's octet count
SSRC_1 (SSRC of first source)
report block 1
fraction lost
cumulative number of packets lost
extended highest sequence number received
interarrival jitter
last SR (LSR)
delay since last SR (DLSR)
SSRC_2 (SSRC of second source)
report block 2
...
profile-specific extensions

Figure 2-8: Sender report RTCP packet [7]



V
P
RC
PT=RR=201
length
header
SSRC of sender
SSRC_1 (SSRC of first source)
report block 1
fraction lost
cumulative number of packets lost
extended highest sequence number received
interarrival jitter
last SR (LSR)
delay since last SR (DLSR)
SSRC_2 (SSRC of second source)
report block 2
...
profile-specific extensions

Figure 2-9: Receiver report RTCP packet [7]

Each RTCP packet begins with a fixed part similar to that of RTP data packets, followed by structured elements that may be of variable length according to the packet type but always end on a 32-bit boundary. The alignment requirement and a length field in the fixed part are included to make RTCP packets “stackable”. Multiple RTCP packets may be concatenated without any intervening separators to form a compound RTCP packet that is sent in a single packet of the lower layer protocol, for example UDP. All RTCP packets must be sent in a compound packet of at least two individual packets, with the following format recommended:

- Encryption prefix:
  If and only if the compound packet is to be encrypted, it is prefixed by a random 32-bit quantity redrawn for every compound packet transmitted.
- SR or RR:
  The first RTCP packet in the compound packet must always be a report packet to facilitate header validation. This is true even if no data has been sent nor received, in which case an empty RR is sent, and even if the only other RTCP packet in the compound packet is BYE.
- Additional RRs:
  If the number of sources for which reception statistics are being reported exceeds 31, the number that will fit into one SR or RR packet, then additional RR packets should follow the initial report packet.

- SDES:
  An SDES packet containing a CNAME item must be included in each compound RTCP packet. Other source description items may optionally be included if required by particular application, subject to bandwith constraints.
- BYE or APP:
  Other RTCP packet types, including those yet to be defined, may follow in any order, except that BYE should be the last packet sent with a given SSRC/CSRC. Packet types may appear more than once.

If the overall length of a compound packet would exceed the maximum transmission unit (MTU) of the network path, it may be segmented into multiple shorter compound packets to be transmitted in separate packets of the underlying protocol.

2.6.6. H.323

This recommendation defines the components, procedures, and protocols necessary to provide audiovisual communications on local area networks. H.323 is applicable to any packet-switched network regardless of the underlying physical layer. The network is expected to provide a reliable delivery mechanism (such as TCP) and unreliable delivery mechanism (such as UDP). An example of such a network is Ethernet using TCP/IP or IPX/SPX protocol stacks. The recommendation is independent of network topology, and H.323 terminals can communicate through hubs, routers, bridges, and dial-up connections via star topologies and multidrop topologies.

H.323 provides various levels of multimedia communications. These levels include voice only, voice and video, voice and data, or voice, video, and data communications over a local area network [8].

2.6.6.1. H.323 terminal

The H.323 terminal provides real-time bidirectional audio, video, and data communications. The recommendation defines call signaling, control messages, multiplexing, audio codecs, video codecs, and data protocols. Figure 2-10 shows an example of an H.323 terminal. H.323 does not specify audio or video equipment, data applications, or the network interface; however, it does mandate certain capabilities in order to provide a minimum level of interoperability. H.225.0 specifies the messages for call signaling, registration, and admissions, as well as packetization and synchronization of the media streams. H.245 specifies the messages used for capability exchange, opening and closing logical channels for media streams, and other commands, requests, and indications.

H.323 provides a variety of media coding options. For audio, G.711, G.728, G.723.1, and G.729 are available. These algorithms provide a choice of lower bit rates, lower delay, or improved quality. For video, H.261 QCIF and CIF as well as all modes of H.263 are available. The choice of audio and video algorithms also provides compatibility with other terminal types, thus eliminating the need to transcode media streams in the gateway. G.711 audio is mandatory; if video is supported, QCIF is mandatory. Receive path delay is optional for both audio and video streams; this may be used to provide lip synchronization and/or jitter control.

Unlike the other ITU-T terminal recommendations, H.323 describes not only terminals, but also a number of other components on the LAN. This includes the gateway, gatekeeper, multipoint controller, multipoint processor, and multipoint control unit.

2.6.6.2. Multipoint conferencing

One of the major differences between H.323 terminals and other ITU-T terminal types is in multipoint conferencing. H.323 has defined several conferencing modes. Point-to-point conferences take place between two terminals, multipoint conferences take place between three or more terminals, and broadcast conferences take place between one sending terminal and many receiving terminals.

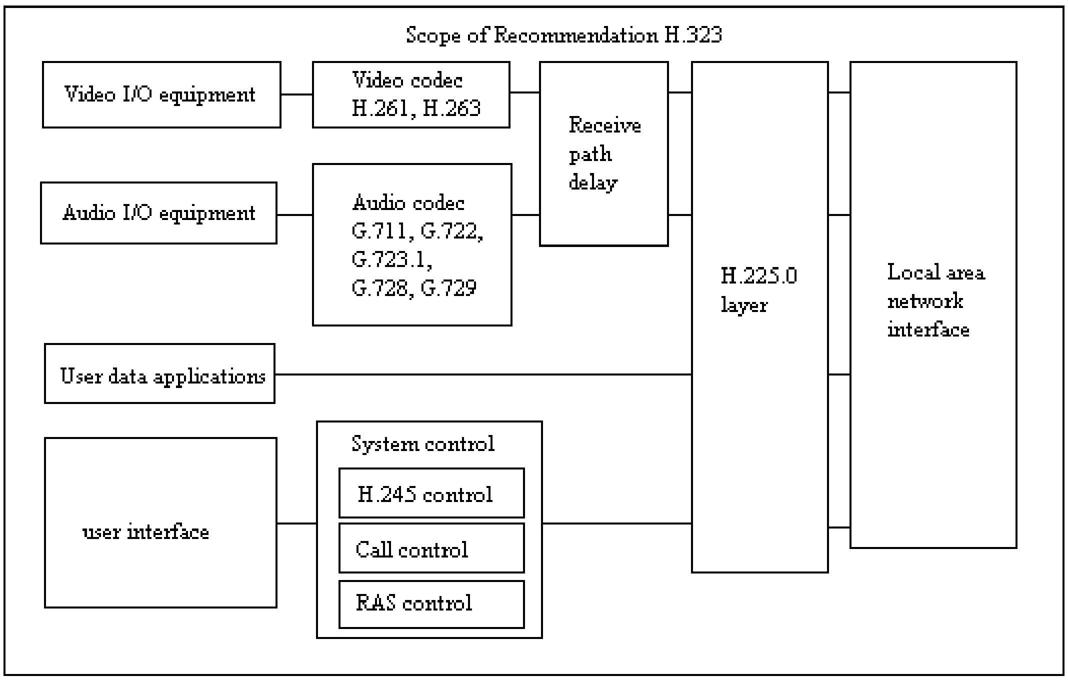


Figure 2-10: H.323 terminal equipment

Three types of multipoint conferences are defined: centralized, decentralized, and hybrid. The centralized multipoint conference uses a multipoint control unit (MCU) to distribute the media streams. Each terminal sends its media streams to the MCU, which then distributes selected or mixed media streams back to the terminals.

2.6.6.3. Gateways

The gateway provides translation of call signaling, control channel messages, and multiplexing techniques between the H.323 terminal and the other ITU-T terminal types. Audio and video transcoding may not be required if both terminal types can find a common communications mode.

2.6.6.4. Gatekeeper

The gatekeeper provides several functions. First, it provides a mechanism for network administrators to control the amount of video telephony traffic on the network. This is done through admission control. Terminals must get permission from the gatekeeper to place or accept a call. The gatekeeper also provides address translation services. This function converts external (telephone numbers) addresses and alias (name) addresses to network addresses, allowing users to maintain the same telephone numbers or alias addresses regardless of changes to their network addresses.

The gatekeeper is optional in H.323 systems, and systems that do not have gatekeepers may not have these capabilities.

2.7. Voice source characterization

The standard coding methods of a voice source result in more or less what is depicted in Figure 2-11. The packet length is dependent of the codec used, sampling frequency, frame length and the number of coded frames in one packet. Frame length indicates the number of samples per coded frame. For example, if we are sampling 8000 samples/s and the frame length is 20 ms we have 160 samples in a frame. The packetization interval is constant and typically 1-2 frame lengths, i.e., 20-40 ms.

A single voice source can be represented by a two-state process. Human speech consists of alternating intervals of inactivity (silence) and activity (talkspurt). The talkspurt lengths average from 0,4 to 1,2 s, and the silence average 0,6 to 1,8 s. This phenomenon has been used for a long time in analog telephony to multiplex and pack multiple calls into one trunk in systems called time-assigned speech interpolators, TASIs, and the digital telephony counterparts, DSIs [1].

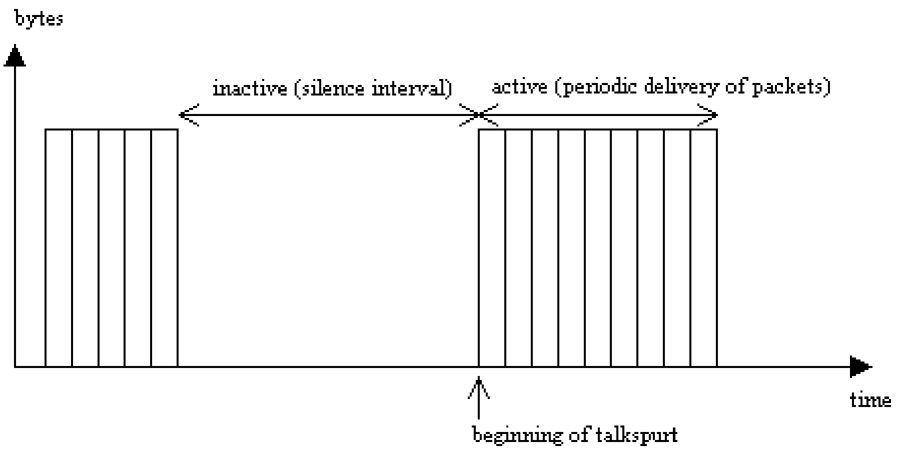

bytes
inactive (silence interval)
active (periodic delivery of packets)
time
beginning of talkspurt

Figure 2-11: The traffic characteristics of a voice stream

Whether the block is to be treated as silence or as part of a talkspurt, is decided by a silence detector. The silence detector does not aim to minimize the talk activity and should kick in only between longer talkperiods (speaker alternation) rather than between words and most sentences. Its main purpose is to suppress background noise and reduce network load during multi-participant conferences [9].

2.7.1. Packetization Process

Implementations of VoIP terminals may vary, but they all do the following:

1. Audio from microphone or line input is A/D converted at the audio input device;
2. The samples are copied to a memory buffer in blocks of frame length;
3. The VoIP application estimates the energy levels of the block of samples;
4. Silence detector decides whether the block is to be treated as silence or as part of a talkspurt;
5. If the block is a talkspurt it is coded with the selected algorithm (e.g., GSM 06.10);
6. Information indicating the packet’s position within the talkspurt is added;
7. A chosen number of blocks of audio are taken to create one RTP packet, and RTP headers are added;
8. The packet is written to correct socket interface (UDP port), IP-headers are added, physical framing and transmission;
9. Packet is received, de-framed, IP-header checked;
10. Packet is read through the UDP socket;
11. RTP-headers are checked for type of payload data, sequence number, timestamp;
12. Sequence number and timestamp are used to detect reordering and duplicates;
13. The insertion point of the incoming audio data is determined in the playout buffer;
14. The block of audio is decoded into samples and inserted in the playout buffer;
15. The block of samples is copied from the buffer to the audio output device;
16. The audio output device D/A converts the samples and outputs them.

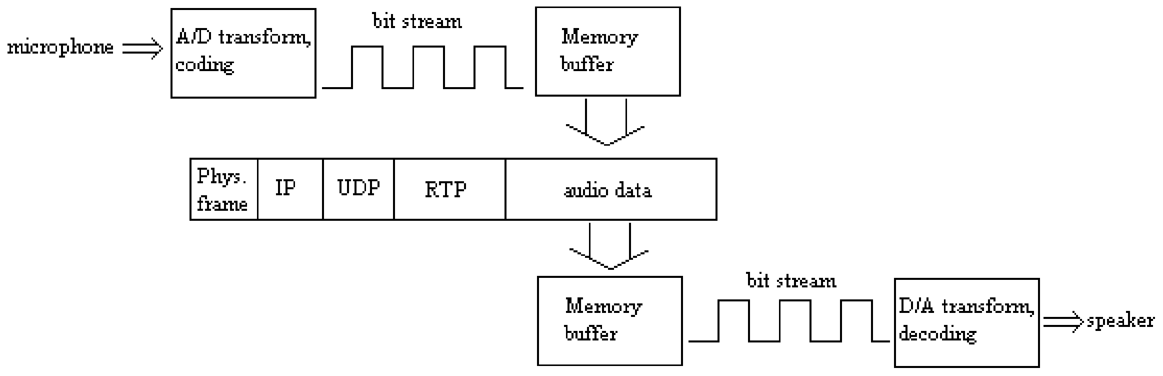


Figure 2-12: Packetization and de-packetization

The packetization and de-packetization is depicted at a very high level in Figure 2-12. In addition to the functions mentioned above the terminal sends congestion information and receives feedback information. Also an automatic gain and echo cancellation can be implemented in the audio input by the VoIP software. The buffering can take place both after sampling and after coding at the sender. The receiver can buffer both after receiving and before D/A-coding [1].

3. SPEECH CODING

Speech coding is conversion of a speech signal into a digital form. The simplest way to do this is by applying sampling theorem directly. This means sampling the waveform at a rate of twice the highest frequency present in the signal, and then digitizing the resulting samples to some desired degree of accuracy. The telephony nominal bandwith is 4 kHz, so the speech signals need to be sampled at a rate of 8000 samples per second. The desired signal-to-noise ratio is dependent on the number of used encoding amplitude levels.

One of the goals of speech coding is to reduce the bit rate. To do this speech waveform specific properties have to be exploited. Adaptive quantizers vary their characteristics over time. This is to match the dynamic range variations of the speech signal. Time-varying filters exploit the short-term and long-term correlations of the signal. Coding methods can also take advantage of the property that, in human hearing, noise can be masked by the speech signal, if the spectral level of the noise is below the spectral level of the speech.

A typical speech coder consists of two modules: an analysis module and a synthesis module. The analysis module extracts from the speech waveform the time varying excitation waveform and the time varying filter parameters. The synthesis module recreates the perceptually best match to the original speech waveform.

This chapter presents the main aspects of speech coding: its history, speech coder attributes, and speech coding techiques. Also the most common speech coders concerning IP voice are presented.

3.1. The history of speech coding

Prior to the era of digital communications, speech was transmitted and stored as an analog signal. Today it can be represented in a digital form, which allows storing and transmitting in a more efficient way. The first speech coder, Homer Dudley’s vocoder, was created almost 60 years ago. It was put to use for the first secure telephony during World War II. From then until the early 1970s, it seemed like only the military was interested in speech coding. All of this changed in the next two decades. The telephone networks of the world became digital. Pulse code modulation at 64 kb/s (PCM), designed to transmit telephone bandwith speech, made it possible to maintain uniform quality for long distance connections. Network operators soon realized that by using 32 kb/s adaptive differential PCM (ADPCM), they could double the capacity of important narrow bandwith links, such as undersea cables. By the 1980s we were entering the era of the PC and the cellular phone. Many new applications, such as digital cellular telephony, voice messaging, videophones, multimedia documents, and Internet telephony, need digital speech coders. Each of these applications has it’s own requirements. Consequently, many new coders were standardized in the 10-year period 1987-1996 [10].

3.2. Speech coder attributes

Speech coding attributes can be divided into four categories: bit rate, delay, complexity, and quality. The applications engineer determines which attributes are most important. It is possible to relax requirements for the less important attributes so that more important ones can be met. Table 3-1 is a list of standardized speech coders and their attributes [10], [11].

Table 3-1: List of standardized speech coders

Standard	*Algorithm*	*Complexity (MIPS)*	*Frame size/looka-head (ms)*	*Compres-sion*	*Bit rate (kb/s)*	*MOS*	*Year finalized*
G.711	PCM	0	0,125/0	1	64	4,10	1972
G.726, G.727	ADPCM	1	0,125/0	4/2,7/2/1,6	16/24/32/40	3,85	1990
G.722			0,125/1,5	1,3/1,1/1	48/56/64		1988
G.728	LD-CELP	30	0,625/0	4	16	3,61	1992, 1994
G.729	CS-ACELP	20	10/5	8	8	3,92	1995
G.729A	CS-ACELP	11	10/5	8	8	3,7	1995
G.723.1	MPC-MLQ	16	30/7,5	10,2/12,1	6,3/5,3	3,9	1995
GSM 06.10	RPE-LTP	10	20/0	4,9	13	3,50	1987
IS-54	VSELP	24	20/5	8	8	3,54	1990
PDC	VSELP		20/5	9,6	6,7		1990
IS-96	QCELP		20/5	7,5/16/32/80	8,5/4/2/0,8		1993
PDC	PSI-CELP		40/10	18,6	3,45		1993
FS-1016	CELP	30		13,3	4,8	3,0	
FS-1015	LPC10E	15		26,7	2,4	2,4	

3.2.1. Bit rate

Bit rate is an attribute that often comes to mind first when thinking of speech coders. The range of bit rates that have been standardized is from 2,4 kb/s for secure telephony to 64 kb/s G.711 PCM and the G.722 wideband (7 kHz) speech coder. Nominal bit rate for a speech coder is the peak rate. Any of the fixed-rate speech coders can be combined with a voice activity detector and made into a simple two-state variable-bit-rate-system. The lower rate could be either zero or some low rate needed to characterize a slowly changing background noise characteristic. Either way, the bandwith of the communications channel is only used for active speech. For circuit multiplication equipment, planners generally assume that individual talkers only have active speech about 40 percent of the time in a two-way conversation. (The percentage is even lower for conference calls.) As a result, the effective bandwith of the link can be increased up to 2,5 times by speech interpolation [10].

3.2.2. Delay

The delay of a speech coder can have a great impact on its suitability for a particular application. Let us compare two different types of applications to illustrate this. The first is a speech coder used for a real-time conversation; the second is a multimedia storage application. Psychologists who have studied conversational dynamics know that if the one way delay of a conversation is greater than 400 ms, the conversation will become more like a half-duplex or push-to-talk experience, rather than an ordinary conversation. In contrast, if a speech or audio file is being downloaded to a client terminal, whether it is delayed an additional 400 ms before starting will be virtually imperceptible to the user. The user is already prepared to wait several seconds between issuing the command and beginning to listen to the file. Thus, a conversation is an example of an application that is most delay-sensitive, while storage is least delay-sensitive.

Reviewing our list in Table 3-1, we find that the highest-rate speech coders, such as G.711 PCM and G.726 ADPCM, have the lowest delay. To achieve higher degrees of compression, speech must be divided into blocks of frames and then encoded a frame at a time. For G.728 the frames are five samples (0,625 ms) long. For first-generation cellular coders, the frames are 20 ms long. This does not account for the full delay, however. The components of the total system delay include the frame size, lookahead, multiplexing delay, processing delay for computation, and transmission delay. The algorithm used for the speech coder will determine the frame size and lookahead. Lookahead means that a corresponding number of samples are needed from the future speech frame. Algorithm’s complexity will have an impact on the processing delay. The system will determine the multiplexing and transmission delays [10].

3.2.3. Complexity

Most speech coders are impemented on DSP’s or other special-purpose hardware. However, recent multimedia speech coders have been implemented on the host CPU of personal computers and workstations. The measures of complexity for a DSP and a CPU are somewhat different, due to the natures of these two systems. At the heart of complexity is always the raw number of computational instructions required to implement the speech coder. DSPs from different vendors have different architectures, and consequently different efficiencies in implementing the same coder.

The measure used to indicate the computational complexity is the number of instructions per second required for implementation. This is usually expressed in millions of instructions per second (MIPS). DSPs also have high-speed static random access memory (RAM) on-chip for storing variables and data. Usually this is from 1000 to 10000 words of RAM. The amount of RAM required is a second measure of complexity. For the coders in Table 3-1, G.726 and G.727 take less than 100 words of RAM; the other ITU coders range from 2000 to 3000.

Finally, DSPs have on-chip read-only memory (ROM) for storing constants and the program instructions. Required ROM storage is the third measure of compexity. For the ITU coders in Table 3-1, G.726 and G.727 use about 1000 words of ROM, while the others are typically in the range of 10000 words. For an implementation on a PC or workstation, the number of instructions per second is the only relevant measure. General purpose computers have far more RAM (although it is usually slower-speed, less expensive dynamic RAM). Both the RAM and ROM from the DSP implementation will be stored in RAM to run the program on a computer. However, general-purpose computers tend to have less efficient processor architectures for implementing digital signal processing algorithms. Consequently, an algorithm requiring 10 MIPS on a DSP may require far more cycles on a computer [10].

3.2.4. Quality

ITU-T Recommendation P.830 [12] provides guidelines for assessing the subjective performance (i.e., speech quality) of speech codecs. Rec. P.830 uses the test procedures that are defined in Recommendation P.800 [13]; the absolute category rating (ACR) method, and the comparison category rating method. The ACR and CCR methods make use of recorded speech samples that have been processed through a number of test “connections”. The processed material is recorded and these samples are presented to panels of listeners. These listeners then make judgments that are defined by the test procedure (ACR or CCR).

3.2.4.1. Absolute category rating

The absolute categoty rating (ACR) method is, perhaps, the most widely used method for evaluating the subjective performance of speech processing equipment in the global switched telephone network. The results of such evaluations are expressed in terms of a mean opinion score (MOS) for each test condition. Each participant in the evaluation hears a collection of speech samples, rating each one in turn. These ratings typically are made using the “listening quality scale”. This scale uses the following verbal descriptions and associated numerical assignments:

Quality of the speech	*Score*
Excellent	5
Good	4
Fair	3
Poor	2
Bad	1

The numerical representations of the ratings are averaged yielding the mean listening-quality opinion score, or simply MOS.

3.2.4.2. Comparison category rating

The comparison category rating (CCR) method provides a means of comparing the quality of two speech samples. Typically, one of the samples is the original (unprocessed) sample, while the other sample is the same material, having been processed through some condition of interest (e.g., a speech codec.) The results of such evaluations are expressed in terms of a comparision mean opinion score (CMOS) for each test condition.

Quite simply, the listeners hear two speech samples. One is the original material, the other is the same sample that has been processed by a speech codec. The order of presentation is random. Listeners are asked to compare the quality of the second speech sample to that of the first using the following verbal descriptions and associated numerical assignments:

Much better	3
Better	2
Slightly better	1
About the same	0
Slightly worse	-1
Worse	-2
Much worse	-3

In effect, listeners provide two judgments with one response: “Which sample has better quality?” and “By how much?”. The numerical representations of the comparison ratings are averaged yielding the comparison mean opinion score (CMOS). A negative CMOS means that the processed material is judged to have lower quality than that of the original material. A CMOS of zero, then, means that the test condition has the same quality as the original material.

As with other paired-comparison methods, the CCR method is quite sensitive. Seemingly small differences in the speech quality measured in the ACR task may result in fairly large differences in CMOS [11].

3.3. Speech coding techniques

3.3.1. Waveform coders and source coders (vocoders)

A broad class of speech coders is termed waveform coders. As the name implies, these coders essentially strive for facsimile reproduction of the signal waveform. In principle, they are designed to be signal independent, hence they can code equally well a variety of signals: speech, music, tones, voiceband data. They also tend to be robust for a wide range of talker characteristics and for noisy environments. To preserve these advantages with minimal complexity, waveform coders typically aim for moderate economies in transmission bit rate. Waveform coders can be optimized and made more signal-specific for greater coding efficiency. This typically is done by observing statistics of a given signal set, so that the waveform coder yields minimal encoding error for this signal class, (i.e., speech). The tailoring of these coders is thus based on a statistical characterization of speech waveforms, as distinct from parameterization of speech information according to some physical model of the signal.

A second class of speech coders depends upon a parsimonius description of speech using a priori knowledge about how the signal was generated at the source. The idea is that certain physical constraints of the signal generation can be quantified, and turned to advantage in efficiency describing the signal. This implies that the signal must be fitted into a specific (in our case, speech-specific) mold and parameterized accordingly. We refer to coding techiques that exploit constraints of signal generation as “source coders”. Source coders for speech are generally referred to as vocoders (a contraction of the words voice coders).

The traditional speech generation model, that dates from so-called “channel vocoder” days, is the source-system model. The sound generating mechanism (the source) is assumed to be linearly separable from the intelligence-modulating, vocal-tract filter (the system). Other assumptions are that speech sounds are either voiced or unvoiced, and that they are generated either from quasi-periodic vocal-cord sound, or from random sound produced by turbulent airflow. By exceedingly meticulous adjustments of parameters, one can demonstrate speech reproduction with good quality. More generally, however, in actual one-pass analysis/synthesis transmission, vocoders tend to be fragile (in terms of parameters such as voiced/unvoiced decision and pitch values), the performance is often talker dependent, and the output speech has a synthetic (less than natural) quality. These characteristics constitute a ceiling on the performance that vocodrs can achieve. But a virtue of their signal parameterization, vocoders can achieve very high economies in transmission bandwidth.

The boundary between waveform coders and vocoders might be thought of as a sort of middle ground, where the design criterion is neither waveform preservation nor signal modeling. Rather, the guiding principle is the preservation of the short time amplitude spectrum of the speech signal in an auditorily palatable way. This middle ground offers opportunities to combine some advantages of both waveform and source coders [14].

3.3.2. Linear predictive coding of speech

One of the most powerful speech analysis techniques is the method of linear predictive analysis. This method has become the predominant technique for estimating the basic speech parameters, e.g., pitch, formants, spectra, vocal tract area functions, and for representing speech for low bit rate transmission or storage. The importance of this method lies both in its ability to provide extremely accurate estimates of the speech parameters, and in its relative speed of computation.

The basic idea behind linear predictive analysis is that a speech sample can be approximated as a linear combination of past speech samples. By minimizing the sum of the squared differences (over a finite interval) between the actual speech samples and the linearly predicted ones, a unique set of predictor coefficients can be determined. (The predictor coefficients are the weighting coefficients used in the linear combination.)

As applied to speech, the various (often equivalent) formulations of linear prediction analysis have been:

- the covariance method
- the autocorrelation formulation
- the lattice method
- the inverse filter formulation
- the spectral estimation formulation
- the maximum likelihood formulation
- the inner product formulation

The first two are the most often used methods.

3.3.2.1. Basic principles of linear predictive analysis

In the linear predictive analysis, the composite spectrum effects of radiation, vocal tract, and glottal excitation are represented by a time-varying digital filter whose steady-state system function is of the form:

$$H(Z) = \frac{S(z)}{U(z)} = \frac{G}{1 - \sum_{k=1}^{p} a_k z^{-k}} \tag{3.1}$$

This system is excited by an impulse train for voiced speech or a random noise sequence for unvoiced speech. Thus, the parameters of this model are: voiced/unvoiced classification, pitch period for voiced speech, gain parameter *G*, and the coefficients $\{a_k\}$ of the digital filter. These parameters, of course, all vary slowly with time.

Figure 3-1 presents block diagram of this simplified model for speech production. The speech samples *s*(*n*) are related to the excitation *u*(*n*) by the simple difference equation:

$$s(n) = \sum_{k=1}^{p} a_k s(n-k) + Gu(n) \tag{3.2}$$

A linear predictor with prediction coefficients, α_k is defined as a system whose output is:

$$\tilde{s}(n) = \sum_{k=1}^{p} \alpha_k s(n-k) \tag{3.3}$$

The prediction error, *e*(*n*), is defined as:

$$e(n) = s(n) - \tilde{s}(n) = s(n) - \sum_{k=1}^{p} \alpha_k s(n-k) \tag{3.4}$$

From Eq. (3.4) it can be seen that the prediction error sequence is the output of a system whose transfer function is:

$$A(z) = 1 - \sum_{k=1}^{p} \alpha_k z^{-k} \tag{3.5}$$

It can be seen by comparing Eqs. (3.2) and (3.4) that if the speech signal obeys the model of Eq. (3.2) exactly and if $\alpha_k = a_k$, then *e*(*n*) = *Gu*(*n*). Thus, the prediction error filter, *A*(*z*), will be an inverse filter for the system, *H*(*z*), of Eq (3.1), i.e.

$$H(z) = G / A(z) \tag{3.6}$$

The basic problem of linear prediction analysis is to determine a set of predictor coefficients $\{a_k\}$ directly from the speech signal in such a manner as to obtain a good estimate of the spectral properties of the speech signal through the use of (3.6). Because of the time-varying nature of the speech signal the predictor coefficients must be estimated from short segments of the speech signal. The basic approach is to find a set of predictor coefficients that will minimize the mean-squared prediction error over a short segment of the speech waveform. The resulting parameters are then assumed to be the parameters of the system function, *H*(*z*), in the model for speech production [15].

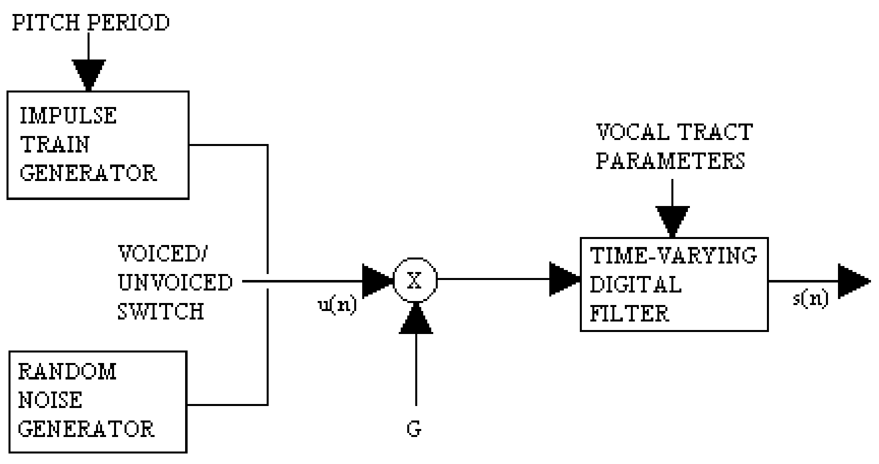


Figure 3-1: Block diagram of simplified model for speech production

3.4. Speech codecs in VoIP products

Most commonly supported speech codecs in VoIP products are G.711 (PCM), G.726 (ADPCM), G.723.1 and G.729. Also GSM 06.10 and various vendor-specific speech codecs are available. Some products support a wide range of speech codecs while some others support only a couple [16].

3.4.1. Pulse code modulation (PCM)

Pulse code modulation (PCM) is essentially analog-to-digital conversion of a special type where the information contained in the instantaneous samples of an analog signal is represented by digital words in a serial bit stream.

If we assume that each of the digital words has n binary digits, there are $M = 2^n$ unique code words that are possible, each code word corresponding to a certain amplitude level. However, each sample value from the analog signal can be any of an infinite number of levels, so that the digital word that represents the amplitude closest to the actual sampled value is used. This is called quantizing. That is, instead of using the exact sample value of the analog waveform $w(kT_s)$, the sample is replaced by the closest allowed value, where there are M allowed values, and each allowed value corresponds to one of the code words.

3.4.1.1. Nonuniform quantizing: µ-Law and A-Law companding

Voice analog signals are more likely to have amplitude values near zero than at the extreme peak values allowed. For example, when digitizing voice signals, if the peak value allowed is 1 V, weak passages may have voltage levels on the order of 0,1 V (20 dB down). For signals, such as these, with nonuniform ampitude distribution, the granular quantizing noise will be a serious problem if the step size is not reduced for amplitude values near zero and increased for extremely large values. This is called nonuniform quantizing since a variable step size is used.

The effect of nonuniform quantizing can be obtained by first passing the analog signal through a compression (nonlinear) ampifier and then into the PCM circuit that uses a uniform quantizer. In the United States a μ-law type of compression characteristic is used. It is defined by:

$$|w_2(t)| = \frac{\ln(1+\mu|w_1(t)|)}{\ln(1+\mu)}, \tag{3.7}$$

where the allowed peak values of $w_1(t)$ are ± 1 and μ is a positive constant that is a parameter. In the United States, Canada and Japan, the telephone companies use μ = 255 compression characteristic in their PCM systems.

Another compression law, used mainly in Europe, is the A-law characteristic. It is defined by:

$$|w_2(t)| = \begin{cases} \dfrac{A|w_1(t)|}{1+\ln A}, 0 \le |w_1(t)| \le \dfrac{1}{A} \\ \dfrac{1+\ln(A|w_1(t)|)}{1+\ln A}, \dfrac{1}{A} \le |w_1(t)| \le 1 \end{cases} \tag{3.8}$$

Where $|w_1(t)| \le 1$ and A is a positive constant. The typical value for A is 87,6. In practice, to approximate A = 87,6 characteristic with steps, let the height of all steps be Δ, then the first 32 steps have step width set to Δ, the next 16 steps have width 2 Δ, followed by 64 steps of width 4 Δ, and, finally, 16 steps of width 64 Δ.

When compression is used at the transmitter, expansion (i.e., decompression) must be used at the receiver output to restore signal levels to their correct relative values. The expandor characteristic is the inverse of the compression characteristic, and the combination of a compressor and an expandor is called a compandor [17].

3.4.1.2. PCM speech codec

Analog voice-frequency signal occupies a band from 300 to 3400 Hz. The minimum sampling frequency would be 2*3,4 = 6,8 kHz. In practice the signal is oversampled and a sampling frequency of 8 kHz is the standard used for voice-frequency signals in telephone communication systems. Each sample is presented with 8 bits thus yielding a bit rate of 64 kbit/s. Then µ- or A-law companding is used. More details can be found in [18].

3.4.2. Adaptive differential PCM (ADPCM)

ITU-T recommendation G.726 defines an ADPCM speech coder with four different bit rates. The available bit rates are: 16, 24, 32 and 40 kbit/s. The different bit rates are achieved by changing the quantizer mode between 2 to 5 bits.

3.4.2.1. ADPCM encoder

Subsequent to the conversion of the A-law or µ-law PCM input signal to uniform PCM, a difference signal is obtained, by subtracting an estimate of the input signal from the input signal itself. An adaptive 31-, 15-, 7-, or 4-level quantizer is used to assign five, four, three or two binary digits, respectively, to the value of the difference signal for transmission to the decoder. An inverse quantizer produces a quantized difference signal from these same five, four, three or two binary digits, respectively. The signal estimate is added to this quantized difference signal to produce the reconstructed version of the input signal. Both the reconstructed signal and the quantized difference signal are operated upon by an adaptive predictor which produces the estimate of the input signal, thereby completing the feedback loop [19].

3.4.3. GSM 06.10 speech codec

Features of the GSM 06.10 speech codec are:

- Bit rate: 13 kbit/s
- Frame length: 160 samples = 20 ms
- 8th-order LPC analysis
- First-order long-term prediction filter

GSM speech codec is based on RPE-LTP (Regular Pulse Exctitation - Long Term Prediction) algorithm.

3.4.3.1. Principle of the RPE-LTP algorithm

The operation of the RPE-LTP algorithm can be divided in three operational parts:

1. Short term prediction

   Speech signal is processed by an 8th-order LPC analysis using autocorrelation method. The residual signal given by LPC analysis is called short term residual.
2. Long term prediction

   In the case of a voiced speech, short term residual is left with an impulse structure. This structure is removed with the long term predictor which is a first-order LPC filter that makes the prediction from the sample that is one pitch-period away.
3. Residual quantization

   The most important feature to enable high compression in LPC-based codecs is the rough quantization of the residual. In RPE-LTP the residual is decimated in the ratio of 1:3. Then the samples are quantized with only 3 bits [20].

3.4.4. G.723.1 speech codec

G.723.1 can be used for compressing the speech or other audio signal components of multimedia services at a very low bit rate as part of the overall H.324 family of standards. This coder has two-bit rates associated with it, 5.3 and 6.3 kbit/s. The higher bit rate has greater quality. The lower bit rate gives good quality and provides system designers with additional flexibility. Both rates are a mandatory part of the encoder and decoder. It is possible to switch between the two rates at any frame boundary.

This coder was optimized to represent speech with high quality at the above rates using a limited amount of complexity. It encodes speech or other audio signals in frames using linear predictive analysis-by-synthesis coding. The excitation signal for the high rate coder is Multipulse Maximum Likelihood Quantization (MP-MLQ) and for the low rate coder is Algebraic-Code-Excited Linear-Prediction (ACELP). The frame size is 30 ms and there is an additional look ahead of 7.5 msec, resulting in a total algorithmic delay of 37.5 msec.

3.4.4.1. Encoder principles

The coder is based on the principles of linear prediction analysis-by-synthesis coding and attempts to minimize a perceptually weighted error signal. The encoder operates on blocks (frames) of 240 samples each, which is equal to 30 msec at the 8 kHz sampling rate. Each block is first high pass filtered to remove the DC component and then divided into four subframes of 60 samples each. For every subframe, a 10th order Linear Prediction Coder (LPC) filter is computed using the unprocessed input signal. The LPC filter for the last subframe is quantized using a Predictive Split Vector Quantizer (PSVQ). The unquantized LPC coefficients are used to construct the short-term perceptual weighting filter, which is used to filter the entire frame and to obtain the perceptually weighted speech signal. For every two subframes (120 samples), the open loop pitch period, L_{OL}, is computed using the weighted speech signal. This pitch estimation is performed on blocks of 120 samples. The pitch period is searched in the range from 18 to 142 samples.

From this point the speech is processed on a 60 samples per subframe basis. Using the estimated pitch period computed previously, a harmonic noise-shaping filter is constructed. The combination of the LPC synthesis filter, the formant perceptual weighting filter, and the harmonic noise-shaping filter is used to create an impulse response. The impulse response is then used for further computations.

Using the pitch period estimation, L_{OL}, and the impulse response, a closed loop pitch predictor is computed. A fifth order pitch predictor is used. The pitch period is computed as a small differential value around the open loop pitch estimate. The contribution of the pitch predictor is then subtracted from the initial target vector. Both the pitch period and the differential value are transmitted to the decoder.

Finally the non-periodic component of the excitation is approximated. For the high bit rate, Multi-pulse Maximum Likelihood Quantization (MP-MLQ) excitation is used, and for the low bit rate, an algebraic-code-excitation (ACELP) is used [21].

3.4.5. G.729 speech codec

The CS-ACELP coder is based on the Code-Excited Linear-Prediction (CELP) coding model. The coder operates on speech frames of 10 ms corresponding to 80 samples at a sampling rate of 8000 samples per second. For every 10 ms frame, the speech signal is analysed to extract the parameters of the CELP model (linear-prediction filter coefficients, adaptive and fixed-codebook indices and gains). These parameters are encoded and transmitted.

At the decoder, these parameters are used to retrieve the excitation and synthesis filter parameters. The speech is reconstructed by filtering this excitation through the short-term synthesis filter. The short-term synthesis filter is based on a 10th order Linear Prediction (LP) filter. The long-term, or pitch synthesis filter is implemented using the so-called adaptive-codebook approach. After computing the reconstructed speech, it is further enhanced by a postfilter.

3.4.5.1. Encoder principles

The input signal is high-pass filtered and scaled in the pre-processing block. The pre-processed signal serves as the input signal for all subsequent analysis. LP analysis is done once per 10 ms frame to compute the LP filter coefficients. These coefficients are converted to Line Spectrum Pairs (LSP) and quantized using predictive two-stage Vector Quantization (VQ) with 18 bits. The excitation signal is chosen by using an analysis-by-synthesis search procedure in which the error between the original and reconstructed speech is minimized according to a perceptually weighted distortion measure. This is done by filtering the error signal with a perceptual weighting filter, whose coefficients are derived from the unquantized LP filter. The amount of perceptual weighting is made adaptive to improve the performance for input signals with a flat frequency-response.

The excitation parameters (fixed and adaptive-codebook parameters) are determined per subframe of 5 ms (40 samples) each. The quantized and unquantized LP filter coefficients are used for the second subframe, while in the first subframe interpolated LP filter coefficients are used (both quantized and unquantized). An open-loop pitch delay is estimated once per 10 ms frame based on the perceptually weighted speech signal. Then the following operations are repeated for each subframe.

The target signal $x(n)$ is computed by filtering the LP residual through the weighted synthesis filter $W(z)/\hat{A}(z)$. The initial states of these filters are updated by filtering the error between LP residual and excitation. This is equivalent to the common approach of subtracting the zero-input response of the weighted synthesis filter from the weighted speech signal. The impulse response $h(n)$ of the weighted synthesis filter is computed. Closed-loop pitch analysis is then done (to find the adaptive-codebook delay and gain), using the target $x(n)$ and impulse response $h(n)$, by searching around the value of the open-loop pitch delay. A fractional pitch delay with 1/3 resolution is used. The pitch delay is encoded with 8 bits in the first subframe and differentially encoded with 5 bits in the second subframe. The target signal $x(n)$ is updated by subtracting the (filtered) adaptive-codebook contribution, and this new target, $x'(n)$, is used in the fixed-codebook search to find the optimum excitation. An algebraic codebook with 17 bits is used for the fixed-codebook excitation. The gains of the adaptive and fixed-codebook contributions are vector quantized with 7 bits, (with MA prediction applied to the fixed-codebook gain). Finally, the filter memories are updated using the determined excitation signal [22].

4. PLAYOUT DELAY ADJUSTMENT

Packet audio tools operate by periodically gathering audio samples generated at the sending host, packetizing them, and transmitting the resulting packet to receiving site(s). For efficiency, the source audio is typically divided into talkspurts (periods of audio activity) and silence periods (periods of audio inactivity, during which no audio packets are generated). In order to faithfully reconstruct the audio at the receiving site, data in packets within a talkspurt must be played out in the same periodic manner in which they were generated.

If the underlying network is free of variations (jitter) in packet delays, a receiving site can simply play out an audio packet as soon as it is received. However, jitter-free, in-order, on-time packet delivery rarely, if ever, occurs in today's packet-switched networks. In order to compensate for these variable delays, a smoothing buffer is thus typically used at a receiver. Received packets are first queued into the smoothing buffer and the periodic playout of packets within a talkspurt is delayed for some amount of time beyond the reception of the first packet in the talkspurt. We refer to this delay as the playout delay of the talkspurt. Clearly, the longer the playout delay, the more likely it is that a packet will have arrived before its scheduled playout time. Excessively long playout delays, however, can significantly impair human conversations. There is thus a critical tradeoff between the length of playout delay and the amount of loss (due to late packet arrival) that is incurred. Generally, delays between talkspurt generation and receiver playout of less than 400 ms [23] and a loss percentage of up to 5% [24] are considered to be quite tolerable in human conversations.

The talkspurt playout delays themselves can be either fixed for the duration of the audio session, or adaptively adjusted. In the Internet, end-to-end delays fluctuate significantly and a constant, non-adaptive, playout delay would thus likely yield unsatisfactory audio quality for interactive audio applications. There are two approaches for adaptive playout adjustment: per-talkspurt and per-packet adjustment. The former uses the same playout delay throughout a talkspurt (and, as a result, faithfully reconstructs the original periodic nature of the received audio data from the sender), but allows different playout delays from one talkspurt to another. While this may result in artificially elongated or compressed silence periods, this is not noticeable in played out speech if the change is reasonably small

[25]. In the latter approach, the playout delay varies from packet to packet. A per-packet adaptive adjustment introduces gaps inside talkspurts and is cited as being damaged to the audio quality [26].

This chapter presents observations in the end-to-end delay characteristics of the Internet that are applied in playout algorithms, the mathematics and formulas needed to evaluate playout algorithms and finally the principles of three different playout algorithms are presented.

4.1. End-to-end delay characteristics

Previous studies [27] have indicated the presence of “spikes” in end-to-end Internet delays. A spike constitutes a sudden, large increase in the end-to-end network delay, followed by a series of packets arriving almost simultaneously, leading to the completion of the spike. Figure 4-1 depicts a typical spike. Each point shown represents a packet arriving at the time indicated by its x-axis value, having experienced an end-to-end network delay equal to the y-axis value [28].

With periodically generated packets, the initial steep rise in the delay spike and the linear, monotonic decrease spike after the initial rise, is due to “probe compression” – the accumulation of a number of packets from the connection under consideration (the audio session, in our case) in a router queue behind a large number of packets from other sources. Probe compression is a plausible conjecture about the cause(s) of delay spikes.

Note that when a delay spike is properly contained within a talkspurt, the next opportunity to change the playout delay (i.e., at the beginning of the next talkspurt) occurs after the delay spike terminates. In such a case, it is not possible to adaptively react to the delay spike, since the delay spike is already over (i.e., the delay has returned to its baseline value) by the next talkspurt and any packets that were so excessively delayed during the delay spike that they missed their playout time have already been lost. In cases where a delay spike spans multiple talkspurts, however, it is advantageous to quickly react to the delay spike. Note also that the “baseline” delays fluctuate less compared to spikes and as a result their delay distribution does not change significantly over time.

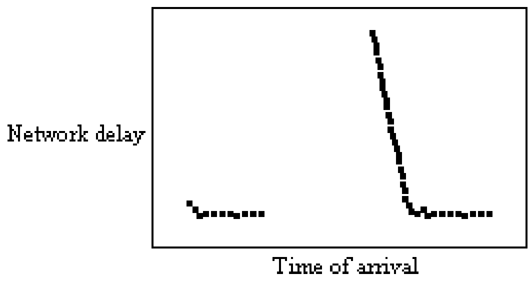


Figure 4-1: A typical spike

4.2. Performance of a playout algorithm

The tradeoff between the average playout delay and loss due to late packet arrival is used as the performance measure in comparing one adaptive playout delay adjustment algorithm with another. Loss and delay are considered on a per-packet rather than per-talkspurt basis for two reasons. First we note that the lengths of talkspurts depend on silence detection algorithms and their parameters. Per-talksurt results are thus closely tied to the silence detection algorithm used. More importantly, different talkspurts have different lengths.

Here a playout delay (or, more accuretely, end-to-end application-to-application delay) is defined to be the difference between the playout time at the receiver and the generation time at the sender. We refer to Figure 4-2 to show the timing information of audio packets and formally define the average playout delay [29].

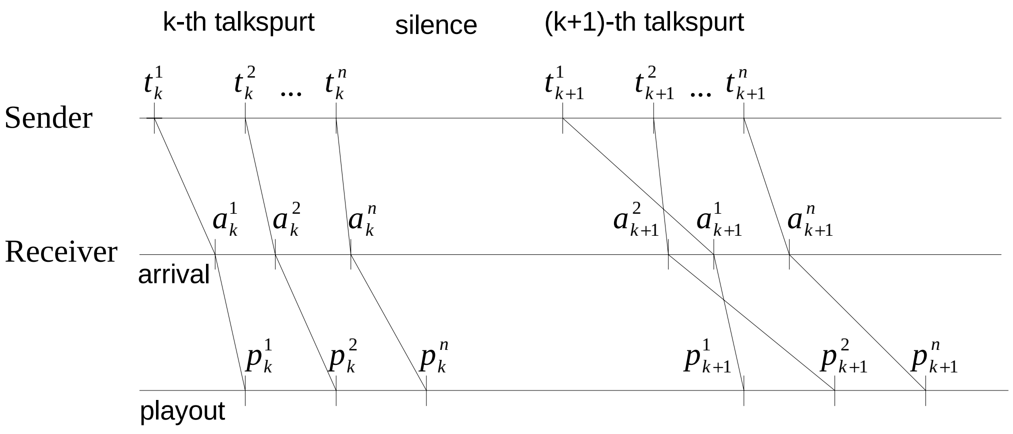

k-th talkspurt
silence
(k+1)-th talkspurt
Sender
t_k^1 t_k^2 ... t_k^n
t_{k+1}^1 t_{k+1}^2 ... t_{k+1}^n
Receiver
arrival
a_k^1 a_k^2 a_k^n
a_{k+1}^2 a_{k+1}^1 a_{k+1}^n
playout
p_k^1 p_k^2 p_k^n
p_{k+1}^1 p_{k+1}^2 p_{k+1}^n

Figure 4-2: Timings associated with the i-th packet in the k-th talkspurt

Consider a trace consisting of *M* talkspurts. We define the following quantities:

- t_k^i : sender timestamp of the i-th packet in the k-th talkspurt
- a_k^i : receiver timestamp of the i-th packet in the k-th talkspurt
- n_k : number of packets in the k-th talkspurt. Here we only consider those packets actually received at the receiver.
- *N*: total number of packets in a trace,

$$N = \sum_{k=1}^{M} n_k \tag{4.1}$$

The playout time of a packet depends on which algorithm is used at the receiver to estimate the playout delay of the packet. Consider a playout algorithm *A*. Then $p_k^i(A)$ is the playout timestamp of the i-th packet in the k-th talkspurt under *A*. If the i-th packet of the k-th talkspurt arrives later than $p_k^i(A)$ (i.e., $p_k^i(A) < a_k^i$), it is considered lost. Otherwise, it is played out with the playout delay of ($p_k^i(A) - t_k^i$). Let $r_k^i(A)$ be an indicator variable for whether the i-th packet of the k-th talkspurt arrives before its playout time, as computed by playout algorithm *A*:

$$r_k^i(A) = \begin{cases} 0, p_k^i(A) < a_k^i \\ 1, otherwise \end{cases} \tag{4.2}$$

The total number of packets played under algorithm *A* is denoted as *N*(*A*) and computed using $r_k^i(A)$:

$$N(A) = \sum_{k=1}^{M} \sum_{i=1}^{n_k} r_k^i(A) \tag{4.3}$$

Then the average playout delay of those played-out packets is defined as:

$$\frac{1}{N(A)} \sum_{k=1}^{M} \sum_{i=1}^{n_k} r_k^i(A)(p_k^i(A) - t_k^i) \tag{4.4}$$

If there are *N* packets in a trace and, among them, *N*(*A*) packets are played out under algorithm *A*, the loss percentage *l* is:

$$l = \frac{N - N(A)}{N} * 100 \tag{4.5}$$

4.3. Some playout algorithms

In this section we present 3 different playout algorithms. Algorithms 1 and 2 are originally reported in [28] and algorithm 3 is suggested in [29]. We introduce the following terminology to be used with these algorithms:

$\hat{d}_k^i$: delay between the generation of the i-th packet of the k-th talkspurt at the sender and its reception at the receiver, namely:

$$\hat{d}_k^i = a_k^i - t_k^i \tag{4.6}$$

We do not need to assume that the sender and receiver clocks are synchronized, but do need to assume that they do not drift. The playout delay of all packets in the k-th talkspurt should be the same due to the periodic nature of packet generation within a talkspurt at the sender and periodic playout at the receiver. Given an algorithm *A*, we denote the playout delay of the k-th talkspurt as $\hat{p}_k(A)$ The playout time of the i-th packet in the k-th talkspurt is then:

$$p_k^i(A) = t_k^i + \hat{p}_k(A) \tag{4.7}$$

4.3.1. Algorithm 1

This algorithm is based on stochastic gradient algorithms used in estimation and control theory [30], and operates by estimating two statistics characterizing the network delay incurred by audio packets: the delay itself, and a variational measure of the observed delays. Each of these estimates is recomputed each time a new packet arrives. The mathematical description of algorithm 1 is presented in Figure 4-3.

$$\boxed{\begin{aligned} &\alpha = 0{,}998002 \\ &\hat{u}_k^i = \alpha * \hat{u}_k^{i-1} + (1-\alpha)\hat{d}_k^i \\ &\hat{v}_k^i = \alpha * \hat{v}_k^{i-1} + (1-\alpha)\,|\,\hat{u}_k^i - \hat{d}_k^i\,| \end{aligned}}$$

Figure 4-3: Algorithm 1

Let $\hat{u}_k^i$ and $\hat{v}_k^i$ be an estimate of the packet delay and variational measure of the i-th packet of the k-th talkspurt. At the beginning of a new talkspurt, the playout delay $\hat{p}_k(A)$ is estimated as follows:

$$\hat{p}_k = \hat{u}_{k-1}^{n_{k-1}} + \beta * \hat{v}_{k-1}^{n_{k-1}} \tag{4.8}$$

Here β is a variation coefficient and provides some slack in playout delay for arriving packets. The larger the coefficient, the more packets that are played out at the expense of longer playout delays. It is thus a parameter that can be used to control the delay/loss tradeoff.

Algorithm 1 is a linear filter that is slow in catching up with a change in delays, but is good at maintaining a steady value, when (1-α), the gain of the estimator, is set to be very low.

4.3.2. Algorithm 2

Algorithm 2 shown in Figure 4-4 has two modes of operation, depending on whether a spike has been detected. In normal mode, it operates like algorithm 1 with a different gain, but in spike-detection mode, $\hat{u}_k^i$ is updated differently.

Algorithm 2 works as follows: In line 2 it is checked if the delay between consecutive packets at the receiver is large enough for it to be called a spike. Once we enter the spike mode on detection of a spike, it seems natural to “follow” the spike. Thus, in spike mode, we allow our estimate to be dictated only by the most recently observed delay values.

The detection of the completion of a spike is a bit tricky. For example, it was observed that in certain cases the delay on completion of the spike was different from the delay before the beginning of the spike. Nonetheless, one prominent characteristics was that a series of packets would arrive one after another almost simultaneously at the receiver, and almost immediately following the observed increase in delay. Since the packets within a talkspurt are transmitted at regular intervals at the sender, near simultaneous arrivals implies that subsequent packets in the burst of arrivals have experienced progressively smaller end-to-end network delays. Thus a variable var is employed with an exponentially decaying value that adjusts to the slope of spike. When this variable has a small enough value, indicating that there is no longer a significant slope, the algorithm reverts back to normal mode.

```
**IF** (mode = = NORMAL)
    **IF** ( $|\hat{d}_k^i - \hat{d}_k^{i-1}| > |\hat{v}_k^{i-1}| * 2 + 800$)
        var = 0;
        mode = SPIKE;
**ELSE**
    var = var/2 + $|(\hat{d}_k^i - \hat{d}_k^{i-1})/8 + (\hat{d}_k^i - \hat{d}_k^{i-2})/8|$;
    **IF** (var ≤ 63)
        Mode = NORMAL;
        $\hat{d}_k^{i-2} = \hat{d}_k^{i-1}$;
        $\hat{d}_k^{i-1} = \hat{d}_k^i$;
        return;
**IF** (mode = = NORMAL)
    $\hat{u}_k^i = 0{,}125 * \hat{d}_k^i + 0{,}875 * \hat{u}_k^{i-1}$;
**ELSE**
    $\hat{u}_k^i = \hat{u}_k^{i-1} + \hat{d}_k^i - \hat{d}_k^{i-1}$;
$\hat{v}_k^i = 0{,}125 * |\hat{d}_k^i - \hat{u}_k^i| + 0{,}875 * \hat{v}_k^{i-1}$;
$\hat{d}_k^{i-2} = \hat{d}_k^{i-1}$;
$\hat{d}_k^{i-1} = \hat{d}_k^i$;
return;
```

Figure 4-4: Algorithm 2

4.3.3. Algorithm 3

The key idea behind this algorithm is to collect statistics on packets that have already arrived and to use them to estimate the playout delay. Instead of using the linear filter mechanism, each packet's delay is logged and the distribution of packet delays is updated at every packet arrival. When a new talkspurt starts, the algorithm calculates a given percentile point *q* in the distribution function of the packet delays for the last *w* packets, and uses it as the playout delay for the new talkspurt. As in algorithm 2, it detects spikes and behaves accordingly: once a spike is detected, it stops collecting packet delays and follows the spike until it detects the end of a spike. Upon detecting the end of a delay spike, it resumes its normal operation.

Algorithm 3 operates in two modes. For every packet that arrives at the receiver, the algorithm checks the current mode and, if necessary, switches its mode to the other in lines 1-7 of Figure 4-5. Lines 9-22 update the delay distribution in normal mode. If a packet arrives with a delay that is larger than some multiple of the current playout delay, the algorithm switches to spike-detection mode. The end of a spike is detected in a similar way: if the delay of a newly arrived packet is less than some multiple of the playout delay before the current spike, the mode is set back to normal. Two parameters head and tail are used in lines 5 and 2 in detecting the beginning and end of a spike.

```
(1)  IF (mode = = SPIKE)
(2)    IF ($\hat{d}_i^k \leq tail$ * old_d)  /* the end of a spike */
(3)      mode = = normal;
(4)  ELSE
(5)    IF ($\hat{d}_k^i > head * \hat{p}_k$)   /* the beginning of a spike */
(6)      mode = SPIKE;
(7)      old_d = $\hat{p}_k$;   /* save $\hat{p}_k$ to detect the end of a spike later */
(8)    ELSE
(9)      IF (delays[curr_pos] ≤ curr_delay)
(10)       count -= 1;
(11)     distr_fcn[delays[curr_pos]] -= 1;
(12)     delays[curr_pos] = $\hat{d}_k^i$;
(13)     curr_pos = (curr_pos+1) % $w$;
(14)     distr_fcn[$\hat{d}_k^i$] += 1;
(15)     IF (delays[curr_pos] < curr_delay)
(16)       count += 1;
(17)     WHILE (count < $w$*$q$)
(18)       curr_delay += unit;
(19)       count += distr_fcn[curr_pos];
(20)     WHILE (count > $w$*$q$)
(21)       curr_delay -= unit;
(22)       count -= distr_fcn[curr_pos];
```

Figure 4-5: Algorithm 3

Depending on the current mode, the playout delay for the next talkspurt is estimated differently in each mode as shown in Figure 4-6. In spike-detection mode, the delay of the first packet of a talkspurt becomes the estimated playout delay for the talkspurt. Otherwise, curr_delay, which is the given percentile point of delay based on previous statistics of packet delays, is used.

```
(1) IF (mode = = SPIKE)
(2)     $\hat{p}_k = \hat{d}_k^1$;
(3) ELSE (mode = = NORMAL)
(4)     $\hat{p}_k = curr\_delay$;
```

Figure 4-6: Playout delay estimation of algorithm 3

The problem with this algorithm is that it takes some time to collect the delay statistics. Until *w* packets are received, playout delays have to be calculated using some other method.

5. OPERATING SYSTEMS AND SCHEDULING

In a workstation, many applications are often running simultaneously. Therefore, operating system and its scheduler is responsible of allocating resources, such as CPU time, to the applications. The used scheduling policy determines how this is done.

In this chapter we present the concepts of operating system and processes. We continue with presenting the most used scheduling policies, and finally the Unix process scheduler is introduced.

5.1. Operating system

Computer software can be roughly divided into two kinds: the system programs, which manage the operation of the computer itself, and the application programs, which solve problems for their users. The most fundamental of all the system programs is the operating system, which controls all the computer’s resources and provides the base upon which the application programs can be written.

A modern computer system consists of one or more processors, some main memory, clocks, terminals, disks, network interfaces, and other input/output devices. All in all, a complex system. Writing programs that keep track of all these components and use them correctly, let alone optimally, is an extremely difficult job. If every programmer had to be concerned with how disk drives work, and with all the dozens of things that could go wrong when reading a disk block, it is unlikely that many programs could be written at all.

Many years ago it became abundantly clear that some way had to be found to shield programmers from the complexity of the hardware. The way that has gradually evolved is to put a layer of software on top of the bare hardware, to manage all parts of the system, and present the user with an interface or virtual machine that is easier to understand and program. This layer of software is the operating system. The situation is shown in Figure 5-1 [31].

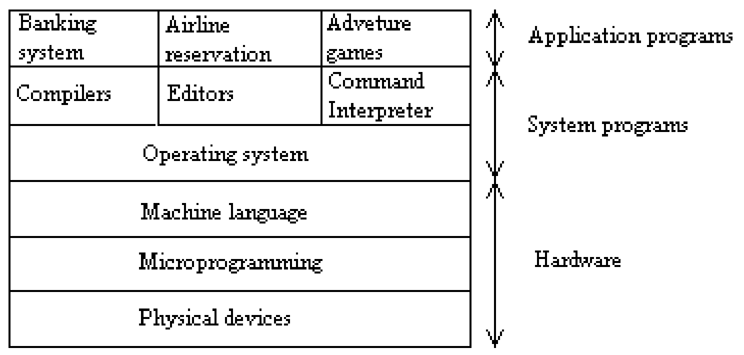


Figure 5-1: A computer system consists of hardware, system programs, and application programs

5.2. Processes

A key concept in all operating systems is the process. A process is basically a program in execution. It consists of the executable program, the program's data and stack, its program counter, stack pointer, and other registers, and all the other information needed to run the program.

Periodically, the operating system decides to stop running one process and start running another, for example, because the first one has had more than its share of CPU time in the past second. When a process is temporaliry suspended like this, it must later be restarted in exactly the same state it had when it was stopped. This means that all information about the process must be explicitly saved somewhere during the suspension. For example, if the process has several files open, the exact position in files where the process was must be recorded somewhere, so that a subsequent READ given after the process is restarted will read the proper data. In many operating systems, all the information about each process, other than the contents of its own address space, is stored in an operating system table called the process table, which is an array (or linked list) of structures, one for each process currently in existence.

Thus, a (suspended) process consists of its address space, usually called the core image, and its process table entry, which contains its registers, among other things [31].

5.3. Process scheduling

When more than one process is runnable, the operating system must decide which one to run first. The part of the operating system concerned with this decision is called the scheduler, and the algorithm it uses is called the scheduling algorithm.
Before looking at specific scheduling algorithms, we should think about what the scheduler is trying to achieve. After all, the scheduler is concerned with deciding on policy, not providing a mechanism. Various criteria come to mind as to what constitutes a good scheduling algorithm. Some of the more obvious possibilities include:

1. Fairness: make sure each process gets its fair share of the CPU.
2. Efficiency: keep the CPU busy 100 percent of the time.
3. Response time: minimize response time for interactive users.
4. Turnaround: minimize the time batch users must wait for output.
5. Throughput: maximize the number of jobs processed per hour.

A little thought will show that some of these goals are contradictory. To minimize response time for interactive users, the scheduler should not run any batch jobs at all. It can be shown that any scheduling algorithm that favors some class of jobs hurts another class of jobs. The amount of CPU time available is finite, after all.

A complication that schedulers have to deal with is that every process is unique and unpredictable. Some spend a lot of time waiting for file I/O, while others would use the CPU for hours at a time if given the chance. When the scheduler starts running some process, it never knows for sure how long it will be until that process blocks, either for I/O, or on a semaphore, or for some other reason. To make sure that no process runs too long, nearly all computers have an electronic timer or clock built in, which causes an interrupt periodically. A frequency of 50 or 60 times a second is common, but on many computers the operating system can set the timer frequency to anything it wants. At each clock interrupt, the operating system gets to run and decide whether the currently running process should be allowed to continue, or whether it has had enough CPU time for the moment and should be suspended to give another process the CPU.

The strategy of allowing processes that are logically runnable to be temporarily suspended is called preemptive scheduling, and in contrast to the run to completion method of the early batch systems. Run to completion is also called nonpreemptive scheduling.

5.3.1. Round robin scheduling

One of the oldest, simpest, fairest, and most widely used algorithms is round robin. Each process is assigned a time interval, called its quantum, which it is allowed to run. If the process is still running at the end of the quantum, the CPU is preempted and given to another process. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks, of course. Round robin is easy to implement. All the scheduler needs to do is maintain a list of runnable processes. When the quantum runs out on a process, it is put at the end of the list.

5.3.2. Priority scheduling

Round robin scheduling makes the implicit assumption that all processes are equally important. Frequently, the people who own and operate computer centers have different ideas on that subject. The need to take external factors into account leads to priority scheduling. The basic idea is straightforward: each process is assigned a priority, and the runnable process with the highest priority is allowed to run.

To prevent high-priority processes from running indefinitely, the scheduler may decrease the priority of the currently running process at each clock tick (i.e., at each clock interrupt). If this action causes its priority to drop below that of the next highest process, a process switch occurs.

Priorities can be assigned to processes statically or dynamically. Priorities can be assigned dynamically by the system to achieve certain system goals. For example, some processes are highly I/O bound and spend most of their time waiting for I/O to complete. Whenever such a process wants the CPU, it should be given the CPU immediately, to let it start its next I/O request, which can then proceed in parallel with another process actually computing. Making the I/O bound process wait a long time for the CPU will just mean having it around occupying memory for an unnecessary long time. A simple algorithm for giving good service to I/O bound processes is to set the priority to $1/f$, where f is the fraction of the last quantum that a process used. A process that used only 2 ms of its 100 ms quantum would get priority 50, while a process that ran 50 ms before blocking would get priority 2, and a process that used the whole quantum would get priority 1.

It is often convenient to group processes into priority classes and use priority scheduling among these classes but round robin scheduling within each class.

5.3.3. Guaranteed scheduling

A completely different approach to scheduling is to make real promises to the user about performance and then live up to them. One promise that is realistic to make and easy to live up is this: If there are *n* users logged in while you are working, you will receive about 1/*n* of the CPU power.

To make good on this promise, the system must keep track of how much CPU time a user has had for all his processes since login, and also how long each user has been logged in. It then computes the amount of CPU each user is entitled to, namely the time since login divided by *n*. Since the amount of CPU time each user has actually had is also known, it is straightforward to compute the ratio of actual CPU had to CPU time entitled. A ratio of 0,5 means that a process has only had half of what it should have had, and a ratio of 2,0 means that a process has had twice as much as it was entitled to. The algorithm is then to run the process with the lowest ratio until its ratio has moved above its closest competitor.

A similar idea can be applied to real-time systems, in which there are absolute deadlines that must be met. Here one looks for the process in greatest danger of missing its deadline, and runs it first. A process that must finish in 10 seconds gets priority over one that must finish in 10 minutes.

5.3.4. Two-level scheduling

If insufficient main memory is available, some of the runnable processes will have to be kept on disk. This situation has major implications for scheduling, since the process switching time to bring in and run a process from disk is orders of magnitude more than switching to a process already in main memory.

A more practical way of dealing with swapped out processes is to use a two-level scheduler. Some subset of the runnable processes is first loaded into main memory. The scheduler then restricts itself to only choosing processes from this subset for a while. Periodically, a higher-level scheduler is invoked to remove processes that have been in memory long enough and to load processes that have been on disk too long. Once the change has been made, the lower-level scheduler again restricts itself to only running processes that are actually in memory. Thus, lower-level scheduler is concerned with making a choice among the runnable processes that are in memory at the moment, while the higher-level scheduler is concerned with shuttling processes back and forth between memory and disk.

Among the criteria that the higher-level scheduler could use to make its decicions are the following ones:

1. How long has it been since the process was swapped in or out?
2. How much CPU time has the process had recently?
3. How big is the process? (Small ones do not get in the way.)
4. How high is the priority of the process?

Again here we could use round robin, priority scheduling, or any of various other methods [31].

5.4. Unix process scheduler

The UNIX system scheduler determines when processes run. It maintains process priorities based on configuration parameters, process behaviour, and user requests; it uses these priorities to assign processes to the CPU.

The SunOS 5.x system gives users absolute control over the order in which certain processes run and the amount of time each process can use the CPU before another process gets a chance. By default, the scheduler uses a time-sharing policy. A time-sharing policy adjusts process priorities dynamically to provide good response time to interactive processes and good throughput to processes that use a lot of CPU time.

The SunOS 5.x system scheduler offers a realtime scheduling policy as well as a time sharing policy. Realtime scheduling allows users to set fixed priorities on a per-process basis. The highest-priority realtime user process always gets the CPU as soon as the process is runnable, even if system processes are runnable. A program can therefore specify the order in which processes run. A program can also be written so that its realtime processes have a guaranteed response time from the system.

For most UNIX environments, the default scheduler configuration works well and no realtime processes are needed. However, when the requirements for a program include strict timing constraints, realtime processes sometimes provide the only way to satisfy those constraints [32].

5.4.1. Scheduling classes

The SunOS 5.x kernel dispatches processes by priority. The scheduler supports the concept of scheduling classes. Classes are defined as realtime (RT), system (SYS), and time-sharing (TS). Each class has a unique scheduling policy for dispatching processes within its class. Figure 5-2 illustrates the concept of classes as viewed by the SunOS 5.x kernel.

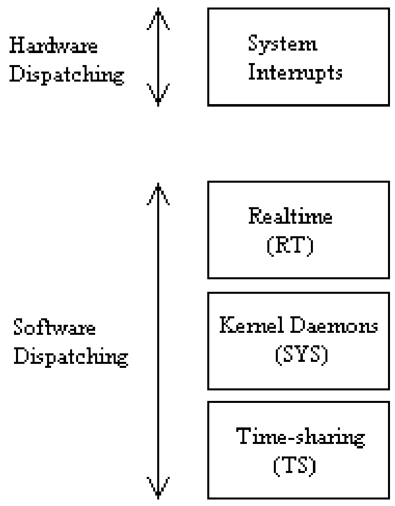


Figure 5-2: Dispatch priorities for scheduling classes

At highest priority are the hardware interrupts; these cannot be controlled by software. The interrupt processing routines are dispatched directly and immediately from interrupts, without regard to the priority of the current process.

Realtime processes have the highest default software priority. Processes in the RT class have a priority and time quantum value. RT processes are scheduled strictly on the basis of these parameters. As long as an RT process is ready to run, no SYS or TS process can run. Fixed priority scheduling allows critical processes to run in a predetermined order until completion. These priorities never change unless an application changes them.

An RT class process inherits the parent’s time quantum, whether finite or infinite. A process with a finite time quantum runs until the time quantum expires or the process terminates, blocks (while waiting for an I/O event) or is preempted by a higher priority runnable real-time process. A process with an infinite time quantum ceases execution only when it terminates, blocks, or is preempted.

The SYS class exists to schedule the execution of special system processes, such as paging, STREAMS, and the swapper. It is not possible to change the class of a process to the SYS class. The SYS class of processes has fixed priorities established by the kernel when the processes are started.

At lowest priority are the time-sharing (TS) processes. TS class processes are scheduled dynamically, with a few hundred milliseconds for each time slice. The TS scheduler switches context in round robin fashion often enough to give every process an equal opportunity to run, depending upon its time slice value, its process history (when the process was last put to sleep), and considerations for CPU utilization. Default time-sharing policy gives larger time slices to processes with lower priority.

Different algorithms dispatch each scheduling class. Class dependent routines are called by the kernel to make decisions about CPU process scheduling. The kernel is class-independent, and takes the highest priority process off its queue. Each class is responsible for calculating a process' priority value for its class. This value is placed into the dispatch priority variable of that process.

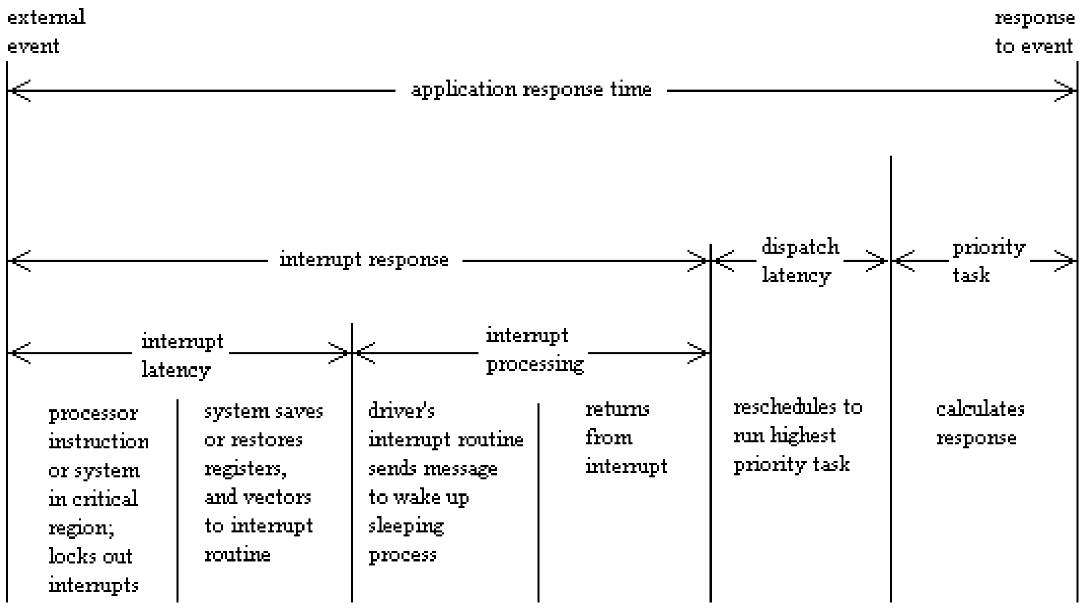
Each class has a set of priority levels that apply to processes in that class. A class-specific mapping maps these priorities into a set of global priorities. It is not required that a set of global scheduling priority maps start with zero, nor that they be contiguous [32].

5.4.2. Dispatch latency

The most significant element in scheduling behaviour for realtime applications is the provision of a realtime scheduling class. The standard time-sharing scheduling class is not suitable for realtime applications because this scheduling class treats every process equally and has a limited notion of priority. Realtime applications require a scheduling class in which process priorities are taken as absolute and are changed only by explicit application operations.

The term dispatch latency describes the amount of time it takes for a system to respond to a request for a process to begin operation. With a scheduler written specifically to honor application priorities, realtime applications can be developed with a bounded dispatch latency. Figure 5-3 illustrates the amount of time it takes an application to respond to a request from an external event.

The overall application response time is composed of the interrupt response time, the dispatch latency, and the time it takes the application itself to determine its response. The interrupt response time for an application includes both the interrupt latency of the system and the device driver's own interrupt processing time. The interrupt latency is determined by the longest interval that the system must run with interrupts disabled; this is minimized in SunOS 5.x using synchronization primitives that do not commonly require a raised processor interrupt level [32].



external event
response to event
application response time
interrupt response
dispatch latency
priority task
interrupt latency
interrupt processing
processor instruction or system in critical region; locks out interrupts
system saves or restores registers, and vectors to interrupt routine
driver's interrupt routine sends message to wake up sleeping process
returns from interrupt
reschedules to run highest priority task
calculates response

Figure 5-3: Application response time

During interrupt processing, the driver's interrupt routine wakes up the high priority process and returns when finished. The system detects that a process with higher priority than the interrupted process is now dispatchable and arranges to dispatch that process. The time to switch context from a lower priority process to a higher priority process is included in the dispatch latency time.

Figure 5-4 illustrates the internal dispatch latency / application response time of a system, defined in terms of the amount of time it takes for a system to respond to an internal event. The dispatch latency of an internal event represents the amount of time required for one process to wake up another higher priority process, and for the system to dispatch the higher priority process.

The application response time is the amount of time it takes for a driver to wake up a higher priority process, have a low priority process release resources, reschedule the higher priority task, calculate the response, and dispatch the task.

With the scheduling techiniques provided with realtime SunOS 5.x, the system dispatch latency is within specified bounds. Tests for dispatch latency and experience with such critical environments as manufacturing and data acquisition have proven that the Sun workstation is an able platform for the development of realtime applications [32].

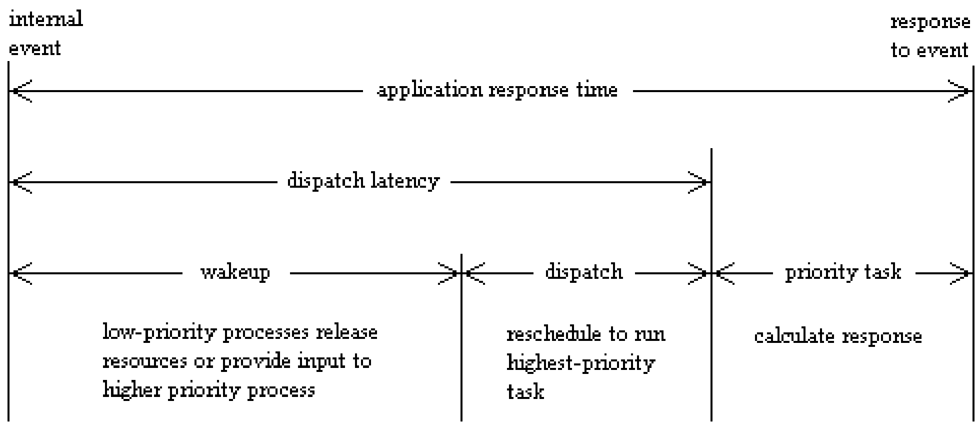

internal event
response to event
application response time
dispatch latency
wakeup
dispatch
priority task
low-priority processes release resources or provide input to higher priority process
reschedule to run highest-priority task
calculate response

Figure 5-4: Internal dispatch latency

6. MEASUREMENTS AND RESULTS

In this chapter we present the experimental research concerning this thesis. The work is divided in three parts. First part handles the processor time consumption in an IP voice terminal, the second part handles the end-to-end delays in an IP voice connection, and in the third part different playout algorithms are compared. The used VoIP client in all experiments is Nevot (Network Voice Terminal) [3]. Nevot was chosen because of its high configurability compared with other VoIP tools. Nevot provides for example a feature to switch off the additional playout delay for testing purposes. Nevot's source codes are freely downloadable in the Internet, which made it possible to compile it from sources and examine the program implementation in practise. Thirdly, Nevot provides a debugging option to record sent and received RTP-headers which is useful for playout algorithm simulations.

6.1. Measurement of the CPU consumption

We wanted to know how much processor time is consumed by Nevot using different audio codecs. In order to do this, version 3.35 of Nevot was compiled from sources using -p flag. This enables us after running the program by using the prof command to produce a profile file which shows for each external text symbol the number of times that function was called and the average amount of time per call.

Nevot provides the following speech codecs:

- PCM	64 kbit/s
- ADPCM	32 kbit/s
- GSM	13 kbit/s
- LPC	4,8 kbit/s

Like presented in Chapter 3, PCM and ADPCM are computationally light codecs compared to GSM and LPC, which are very computationally intensive.

6.1.1. Setup of the measurement

We had two Sun Ultra Enterprise 1 workstations connected to a 10BaseT Ethernet. Configuration of the workstations is presented in Table 6-1.

Table 6-1: Configuration of the workstations

CPUs	1
Clock speed	167 MHz
On chip cache	16 KB I-cache + 16 KB D-cache
External cache	512 KB
SPECint95	5,56
SPECfp95	9,06
Main memory	64 MB
Operating system	SunOS 5.5.1

For comparison, performance values for an upscale PC (Intel AL440LX motherboard (233MHz)) are: SPECint95: 9,47, and SPECfp95: 7,04 [33].

Both workstations were running Nevot version 3.35 for 5 minutes in each measurement. Packet sizes were set to 20 ms and silence detection was disabled so that both clients were continuously sending and receiving packets. Other settings of the program were as following:

Automatic gain control: enabled

- time constant: 10,0 s.
- low volume: -40 dB
- high volume: -10 dB

Echo suppression: enabled

Volume meter: enabled

- update interval: 3 packets
- update threshold: 2,0 dB

Delay adjustment: enabled

- time constant: 10 packets
- initial delay: 0,10 s
- minimum delay: 0,01 s
- maximum delay: 5,00 s
- variance multiplier: 4
- packet late = talkspurt: disabled

Timeouts:

- soft timeout: 4 intervals
- hard timeout: 20 intervals

6.1.2. Results of the measurements

In the first measurement audio coding was set to PCM. Profile file for this measurement is shown in Appendix 1. Total amount of used CPU time is shown to be 3,17 seconds. However, the resolution of the workstation clock sets constrains to this measurement. In Sun Ultra Enterprise1 the resolution of the clock is1 µs. Thus function calls that last less than this are not taken into account at all. We can manually calculate an upper limit for the used CPU time by approximating all function calls that on average are shown to be below one microsecond to 1 µs. This approximation gives additional 0,71 seconds yielding total of 3,88 seconds of used CPU time.

There are also few function calls where the number of function calls and and average time of the call is not shown. These functions belong to some libraries that are not compiled themselves with -p flag and therefore this information is not available. These libraries are libgsm.a, libtk.a and libtcl.a. Some of these functions might also belong to those whose approximate executíon time is below the workstation clock resolution, and are not considered in the error approximation. However, this source of error is obviously relatively small and the calculations performed here should be quite accurate.

We can see that function tx_packet is entered 15190 times during measurement. When we divide total used CPU time with this number, we get the used CPU time per one 20 ms packetization interval which is 0,26 milliseconds. This means that the program is consuming only 0,26/20 = 1,3 percent of the total CPU time using PCM audio coding.

In the next measurement we changed the audio coding to ADPCM. Profile file for this measurement is shown in Appendix 2. Now the total amount of used CPU time is 6,63 seconds. After adding approximated correction of 0, 80 seconds yields 7,43 seconds total used CPU time. When we divide this with the number of sent packets (15221) we get 0,49 ms which is the consumed CPU time per one 20 ms packetization interval. Thus the program is now consuming 0,49/20 = 2,5 percent of the total CPU time.

In the third measurement the audio coding is changed to GSM. Profile file for this measurement is shown in Appendix 3. Similarly as in two earlier measurements, we compute the total used CPU time which is 17,25 seconds with the added correction. Used CPU time per packet is 17,25/15137 = 1,14 ms. The program is now consuming 1,14/20 = 5,7 percent of the total CPU time.

Finally in the fourth measurement the audio coding is changed to LPC. Profile file for this measurement is shown in Appendix 4. Total used CPU time with approximated correction is 27,65 seconds which yields 27,65/15120 = 1,83 ms per packet and is 1,83/20 = 9,1 percent of the total CPU time. The results of these four measurements are summarized in Table 6-2.

Table 6-2: Used CPU time in milliseconds and percent of used CPU time from total CPU time with different audio codings

Used CPU time	*PCM*	*ADPCM*	*GSM*	*LPC*
in milliseconds	0,26	0,49	1,14	1,83
in percent	1,3	2,5	5,7	9,1

These results show that the processing delay increases with the complexity of the codec. However, the used workstations are so powerful that even with an LPC codec the used CPU time stays under 10 % of the total time. In this measurement both workstations were transmitting and receiving packets and the values in Table 6-2 include both coding and decoding in one workstation. Thus the contribution of the processing delays in the end-to-end delay stay under 4 ms with any available codec.

6.2. Measurement of the end-to-end delay

6.2.1. Setup of the measurement

In this measurement we had two Sun Ultra Enterprise1 workstations with SunOS 5.5.1 connected to a 10BaseT Ethernet (see Fig. 6-1). Both workstations were running Nevot version 3.35. Delays were measured with four different audio codings. For each audio coding both half-duplex and full-duplex traffic was measured. We also used two different process priorities for Nevot to see if the operating system has some contribution to the delay. Used priorities were normal time-shared class priority with user priority 0 and realtime priority with the highest possible priority 59 and time slice of 1 second. Each type of measurement was repeated 10 times. Packet size was set to 20 ms in all measurements.

We used a function generator to generate a 1 kHz square wave and fed it to the microphone input of the first workstation. This signal was also fed to the first channel of the oscilloscope. From the other workstation’s headphones output the signal was taken to the second input of the oscilloscope. When the function generator was switched on, it triggered the oscilloscope and the total end-to-end delay could be read from the oscilloscope screen.

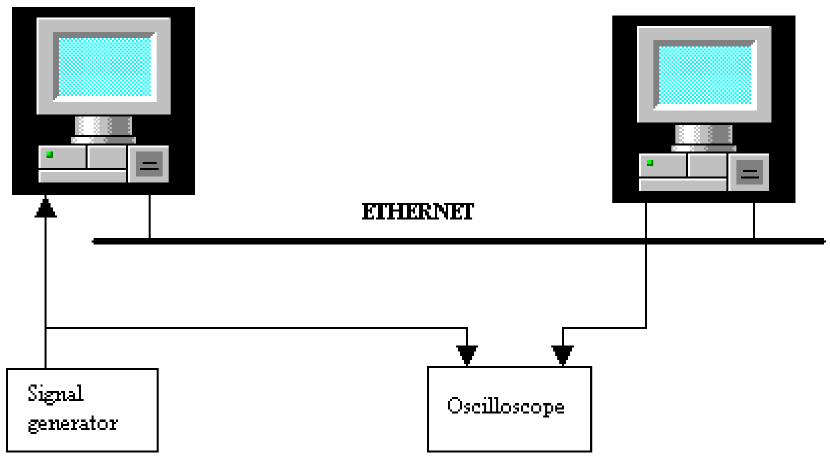


Figure 6-1: Setup of the end-to-end measurement

In the Nevot of the first workstation we had the following settings:

Automatic gain control: enabled

- time constant: 10,0 s.
- low volume: -40 dB
- high volume: -10 dB

Echo suppression: enabled

Silence detection: enabled

- before talkspurt: 0 packets
- after talkspurt: 0 packets
- threshold incrementation: 0.00 dB
- minimum threshold: - 12.0 dB
- maximum threshold: 0.0 dB
- hysteresis: 0.0 dB
- echo suppression: 3.0 dB

Volume meter: enabled

- update interval: 3 packets
- update threshold: 2,0 dB

Delay adjustment: disabled

Timeouts:

- soft timeout: 4 intervals

- hard timeout: 20 intervals

It is important to note that silence detection was enabled and packets before talkspurt were set to 0. This means that we were measuring the delay of the first packet in talkspurt which contains data and no silent packets were sent before it. Also the minimum threshold must be high enough that noise doesn’t trigger the talkspurt and distort the results of the measurement. It is also important that delay adjustment was disabled in this measurement.

In the other workstation, the settings were the same except that when we were measuring full duplex traffic silence detection was off and thus during the measurement both workstations were both sending and receiving packets. During the measurements of half-duplex traffic the other workstation was only receiving packets.

6.2.2. Results of the measurements

Resuls of each used audio coding are presented in Tables 6-3 - 6-6. Results are summarized in Table 6-7.

Table 6-3: End-to-end delays using PCM audio coding

TS, half duplex [ms]	*TS, full duplex [ms]*	*RT, half duplex [ms]*	*RT, full duplex [ms]*
32	34	31	31
33	32	30	31
31	32	31	32
31	32	31	32
32	37	32	31
33	34	31	31
31	31	32	33
34	31	31	32
36	32	33	30
34	32	31	31
32,7	*32,7*	*31,3*	*31,4*

Table 6-4: End-to-end delays using ADPCM audio coding

TS, half duplex [ms]	*TS, full duplex [ms]*	*RT, half duplex [ms]*	*RT, full duplex [ms]*
35	37	32	31
33	37	31	33
33	37	31	31
35	35	34	32
33	34	31	32
32	34	31	32
32	34	32	32
33	36	31	32
32	33	31	33
36	32	32	32
33,4	*34,9*	*31,6*	*32,0*

Table 6-5: End-to-end delays using GSM audio coding

TS, half duplex [ms]	*TS, full duplex [ms]*	*RT, half duplex [ms]*	*RT, full duplex [ms]*
34	33	32	33
34	34	32	33
34	33	32	34
36	33	34	34
38	36	34	33
36	33	32	33
34	34	32	35
34	33	33	34
33	33	32	34
34	33	32	33
34,7	*33,5*	*32,5*	*33,6*

Table 6-6: End-to-end delays using LPC audio coding

TS, half duplex [ms]	*TS, full duplex [ms]*	*RT, half duplex [ms]*	*RT, full duplex [ms]*
34	37	32	34
34	36	32	34
34	36	34	35
36	35	34	36
36	41	33	36
35	37	33	36
37	35	33	36
35	35	32	35
35	35	34	35
34	37	33	35
35,0	*36,4*	*33,0*	*35,2*

Table 6-7: Average end-to-end delays in different audio codings

	TS, half duplex [ms]	*TS, full duplex [ms]*	*RT, half duplex [ms]*	*RT, full duplex [ms]*
PCM	32,7	32,7	31,3	31,4
ADPCM	33,4	34,9	31,6	32,0
GSM	34,7	33,5	32,5	33,6
LPC	35,0	36,4	33,0	35,2

6.2.3. Analysis of the measurements

Measurements were done over a non-loaded LAN and network caused delay was measured with Ping and was shown to be constantly around 0,5 ms. The framing delay of 20 ms is contributed in all results.

Hardware caused delays were measured by directing the signal from microphone input to headphones output by Audio tool program. Thus, a 1 kHz square-wave input signal was A/D converted and then D/A converted in the audio hardware. Input signal was connected to the first channel of an oscilloscope and output signal to the second channel of the oscilloscope. HW delay was measured several times and was constantly 2,2 ms.

If we take for example full duplex measurement with real time priorities and PCM audio coding and subtract 31,4 ms - 20 ms (framing delay) - 2*0,26 ms (processing delay) - 0,5 ms (network delay)- 2,2 (HW delay) we are left with 8,2 milliseconds. This delay is caused by buffering. Buffering delay is introduced in 4 places in the end-to-end delay. In the sending end it takes place both after sampling and after coding. In the receiving end buffering takes place both after receiving and before D/A coding. Small variations in results with RT class processes are caused by varying buffering delays.

Operating system can increase the delay if Nevot is run as a time-sharing class process because then it's not guaranteed to be scheduled at predicted time intervals. The operating system caused delay depends totally from the other processes running simultaneously in the workstations. In these measurements it varied between 0 and 2,9 milliseconds. These measurements were done in the summertime when the workstations were lightly loaded. This explains the small contribution of the operating system caused delay. Components of the end-to-end delay are illustrated in Table 6-8.

Table 6-8: Components of the end-to-end delay

Delay component	*Delay in ms*
Framing delay	20,0
Processing delay	0,5 - 3,7
HW delay	2,2
Network delay	0,5
OS delay	0 – 2,9
Buffering delay	8,2 - 8,8

An interesting thing was noted in the end-to-end delay behaviour with Nevot. When a new talkspurt is started, the delay is first around 30 ms as shown earlier, but within two seconds it increases by 40 ms and after about 10 seconds it increases by another 20 ms. This was found out to be caused by the Nevot that was receiving packets.

When a new talkspurt begins, Nevot starts buffering data in the playout buffer. It periodically checks once in a second that there is certain amount of data in the playout buffer. In the beginning of a talkspurt playout buffer is empty so when the program enters this check, it copies the same packet two times to the playout buffer. When it comes to this point again after one second, it again copies received audio packet twice to the playout buffer. 10 seconds later this is repeated by the third time. This action implicitly increases end-to-end delay by 60 ms. The reason is just to avoid buffer underflows in case of late packets. This behaviour can be clearly seen in oscilloscope screen by measuring end-to-end delay like explained earlier but using a 10 Hz sinus signal as input.

The corresponding lines in the source code can be found from file rx.c:

```
/*
 * Every so often, check queue occupancy. If underflow error,
 * write audio block twice.
 */
if (++underflow > 50) {
  int q = ahw_write_queue(&underflow);
  if (underflow) {
     ring_action(r, r->next, nevot.rx.samples, (ring_action_t *)ahw_write,
     0, 0, nevot.rx.ad.bytes_per_frame);
     debug(DEBUG_AUDIO, "audio underflow (%d)", q);
  }
```

Nevot also provides a debugging option which enables us to record RTP-headers of the received packets. Recorded debugging file is presented in Appendix 5. Corresponding events when audio block is written twice to the playout buffer are seen at timestamps 901004499.820256, 901004500.819931 and 901004510.998093.

We removed this feature by simply replacing the line: “if (++underflow > 50) {“ in the source code with: “if(0) {“ , and compiled Nevot again. The end-to-end delays with and without this correction are illustrated in Figure 6-2. No reduction to sound quality caused by this change was noticed in our informal subjective listening tests where the network was lightly loaded. The evaluation of sound quality with higher network loads and different playout delay settings would require more testing and is not within the scope of this thesis.

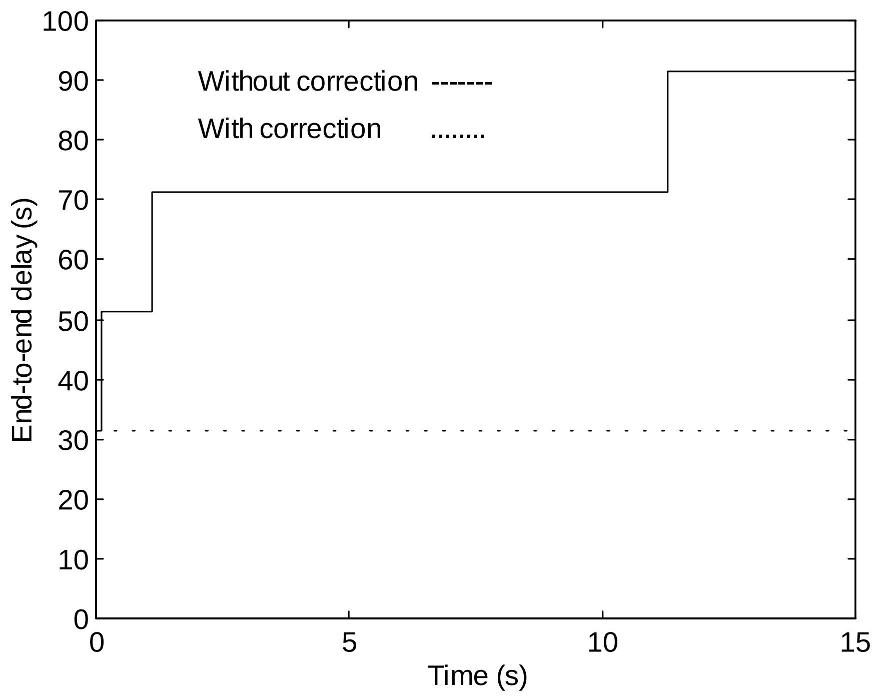


Figure 6-2: End-to-end delays with and without correction

6.3. Comparison of playout algorithms

In this section we compare the performance of the playout algorithms that were presented in Section 4.3. The performance metric we use to compare different playout algorithms is the average playout delay vs. loss percentage. To evaluate algorithms 1-3, we generate some traces which include the sender and received timestamps of each packet from a trace. Using Matlab programs we can simulate the different algorithms. Playout delays for each packet are calculated and we can determine if a packet has arrived before its playout time. Thus we are able to calculate the loss percentage and average playout delay for each algorithm with each trace.

First we compare each algorithm separately using different values for their main parameters. For algorithms 1 and 2 we use different values of α and for algorithm 3 we alternate the window parameter *w*. After this we compare algorithms 1 to 3 with each others using parameters that gave the best performance with each trace. Finally, we will present a new algorithm, referred to as algorithm 4, which is a combination of algorithms 1 and 3. We will then compare the performance of this algorithm with algorithms 1 and 3.

6.3.1. Generation of the traces

In order to simulate playout algorithms, we have to generate some traces which illustrate the delays experienced by voice packets. Setup of this measurement is shown in Figure 6-3. A half-duplex voice connection is established between the two Sun Ultra Enterprise 1 workstations. A 1 kHz sinus signal is suppied by a function generator to the sendind workstation's microphone input. The signal generator is manually switched on/off to generate talkspurts and silent periods. Both workstations are running Nevot 3.35 and transmitted and received RTP-timestamps are recorded to files using the debugging option of Nevot.

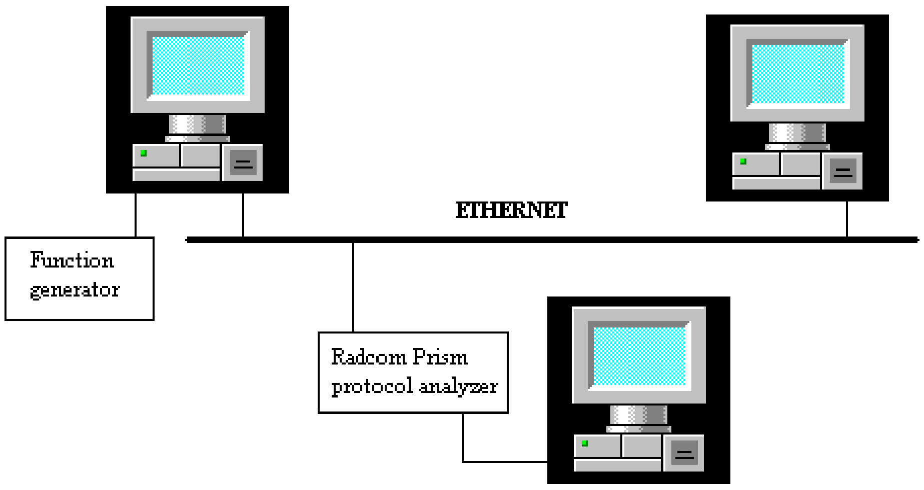


Figure 6-3: Setup for generating the traces

Network load was generated using Radcom Prism 200 protocol analyzer. Loads used with different traces are shown in Table 6-9.

Table 6-9: Used network loads

Trace nr.	*Traffic description*	*Frames/s*	*Bits/frame*	*Load/Mbps*	*Length/packets*
Trace 1	Small packets	3000-5000	160	3,936-6,560	25830
Trace 2	Small packets, high load	2000-3500	320	5,184-9,072	25023
Trace 3	Large packets, bursty load	100-860	1450	1,163-10,000	24048
Trace 4	Variable size packets, bursty load	200-860	160-1450	0,2624-10,000	25337

Packet delays of these traces are shown in Figures 6-4 to 6-7. Figures show only the variable component of the delay, i.e., they are substracted with the minimum delay in the whole trace. This is done because workstation clocks are not synchronized and their clock offsets are several seconds.

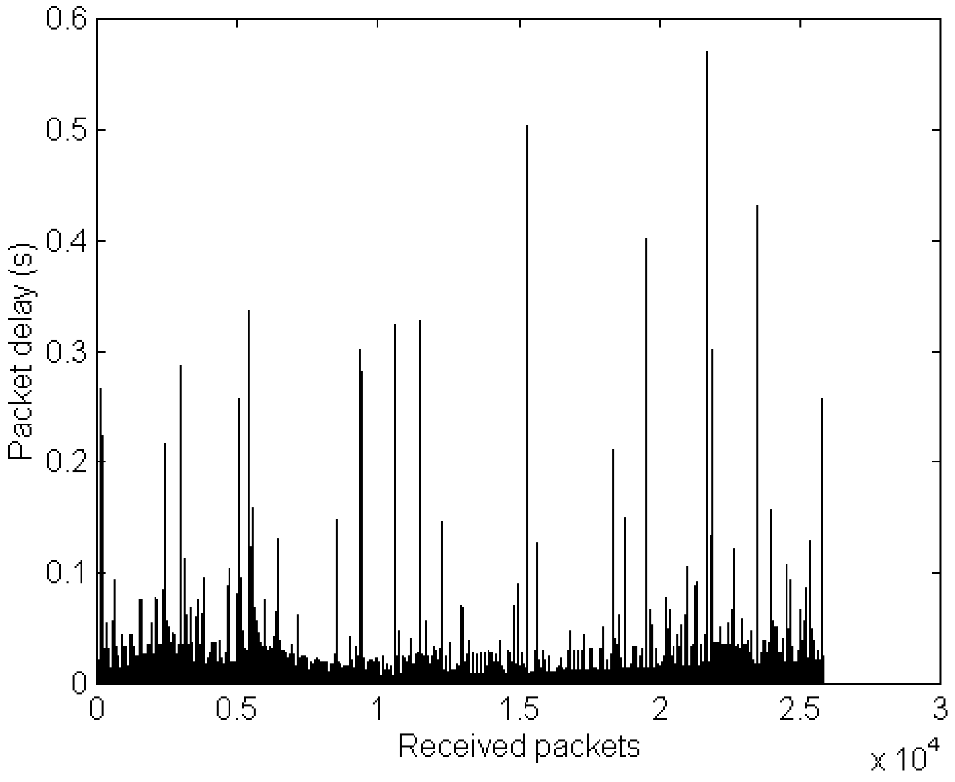


Figure 6-4: Packet delays of trace 1

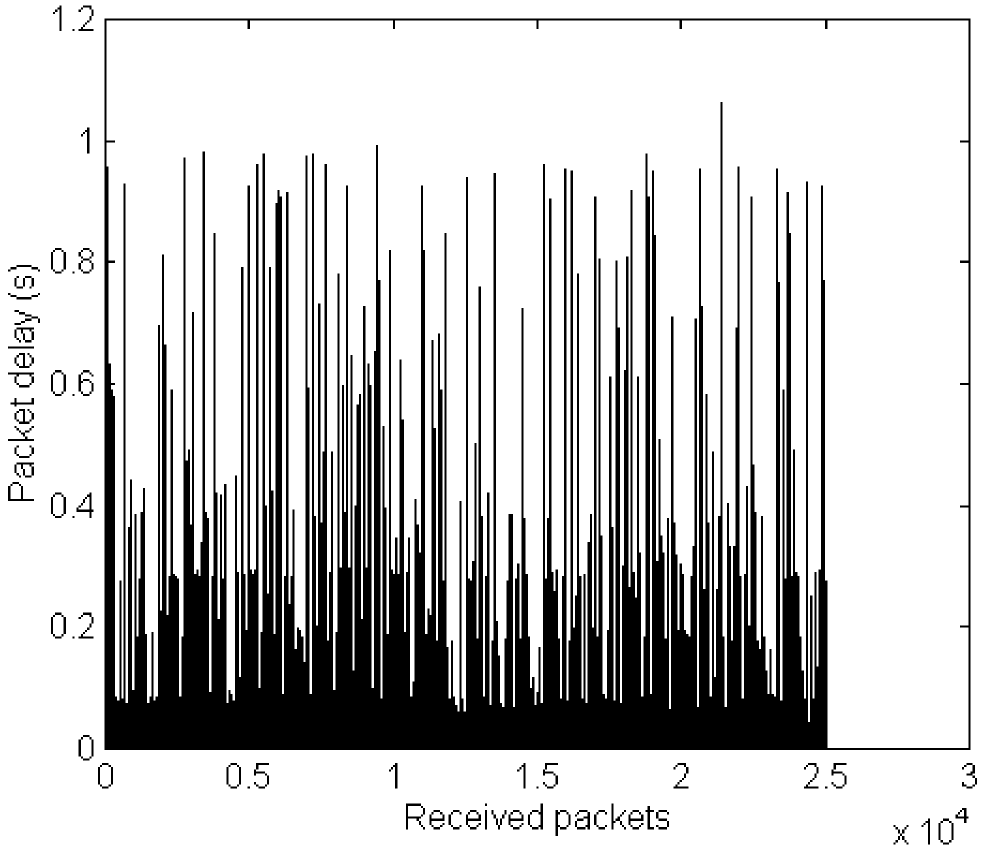


Figure 6-5: Packet delays of trace 2

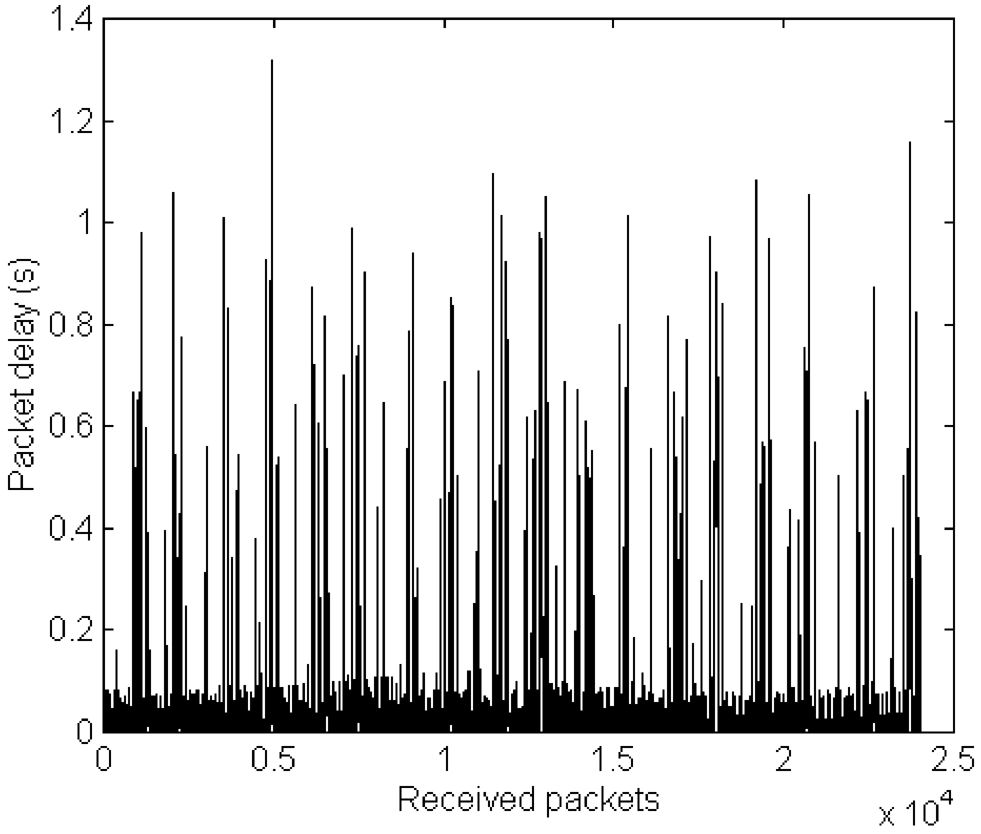


Figure 6-6: Packet delays of trace 3

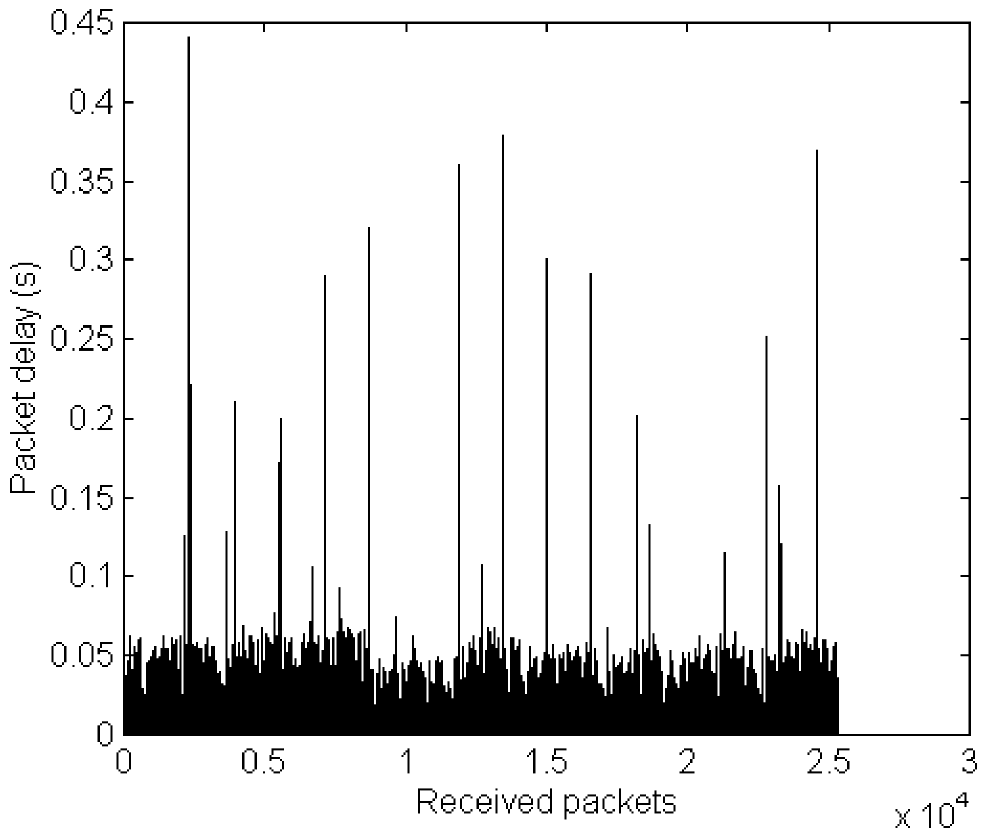


Figure 6-7: Packet delays of trace 4

6.3.2. Performance of algorithm 1

The playout delays for each packet in each trace are calculated using recorded RTP-headers. The trade-off between average playout delay and loss ratio is controlled with different values of β. The value pairs of average playout delay and loss ratio with each value of β are connected by lines to obtain the presented figures.

Values of β are varied from 0,5 to 20 at steps of 0,5. In [28] α is suggested to be 0,998002 for algorithm 1. Here we compare algorithm 1 with α = 0,990, 0,995, 0998002 and 0,999. Figures 6-8 to 6-11 illustrate the performance of algorithm 1 with different values of α. The Matlab code for algorithm 1 is presented in Appendix 6.

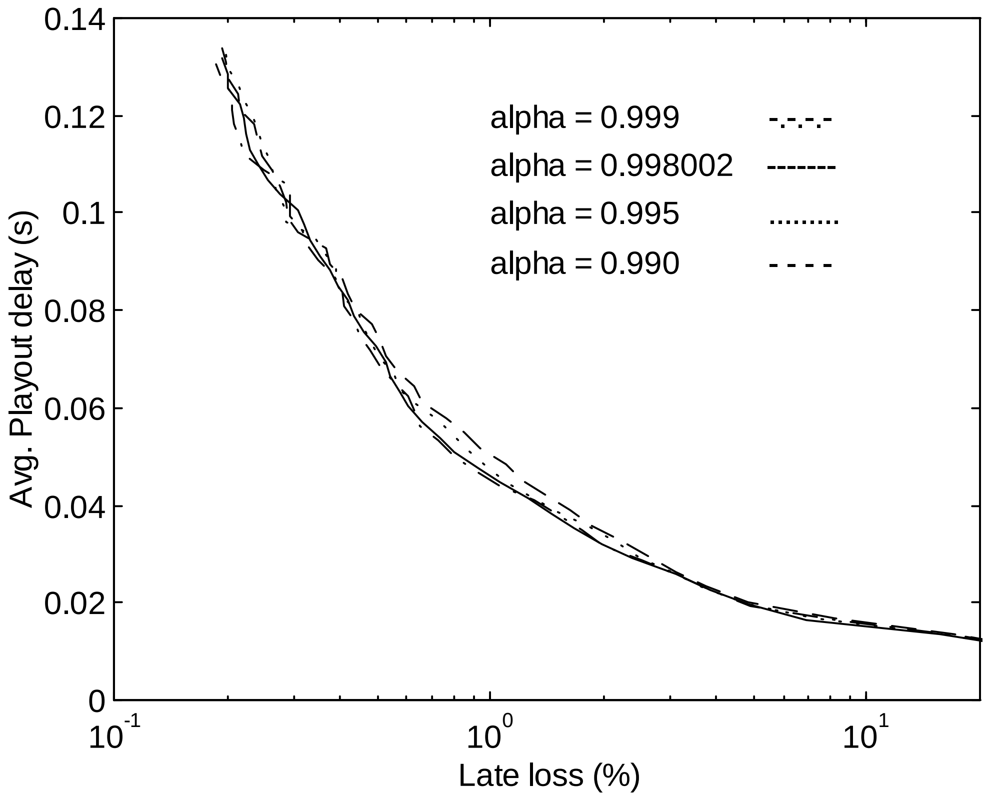


Figure 6-8: Algorithm 1 on trace 1

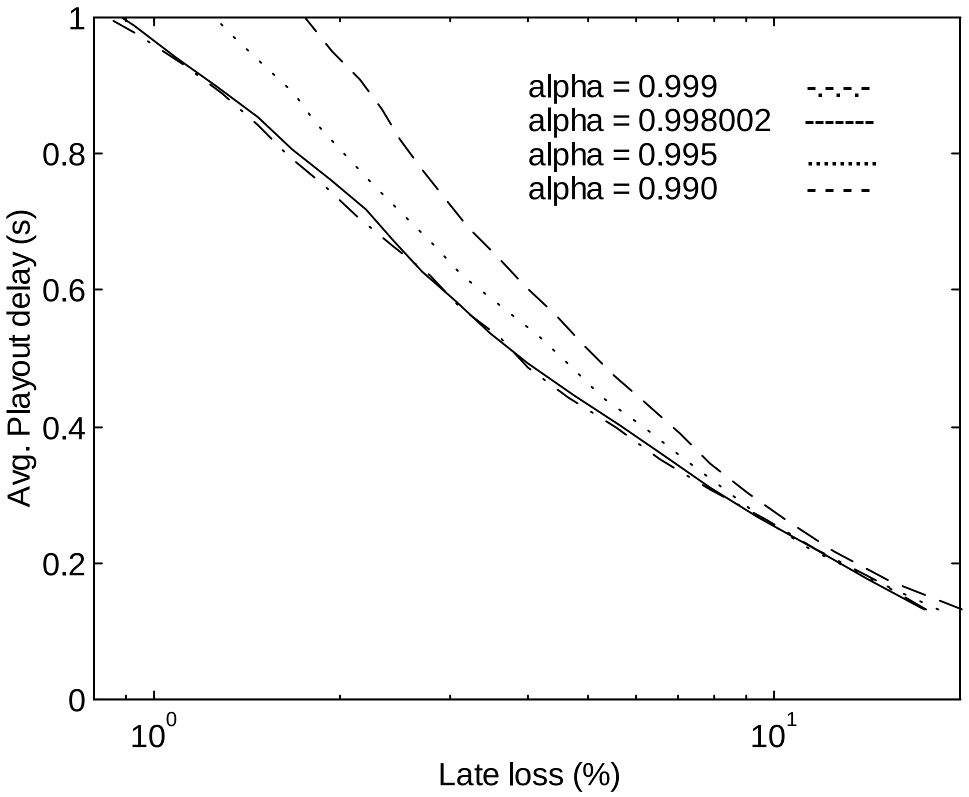


Figure 6-9: Algorithm 1 on trace 2

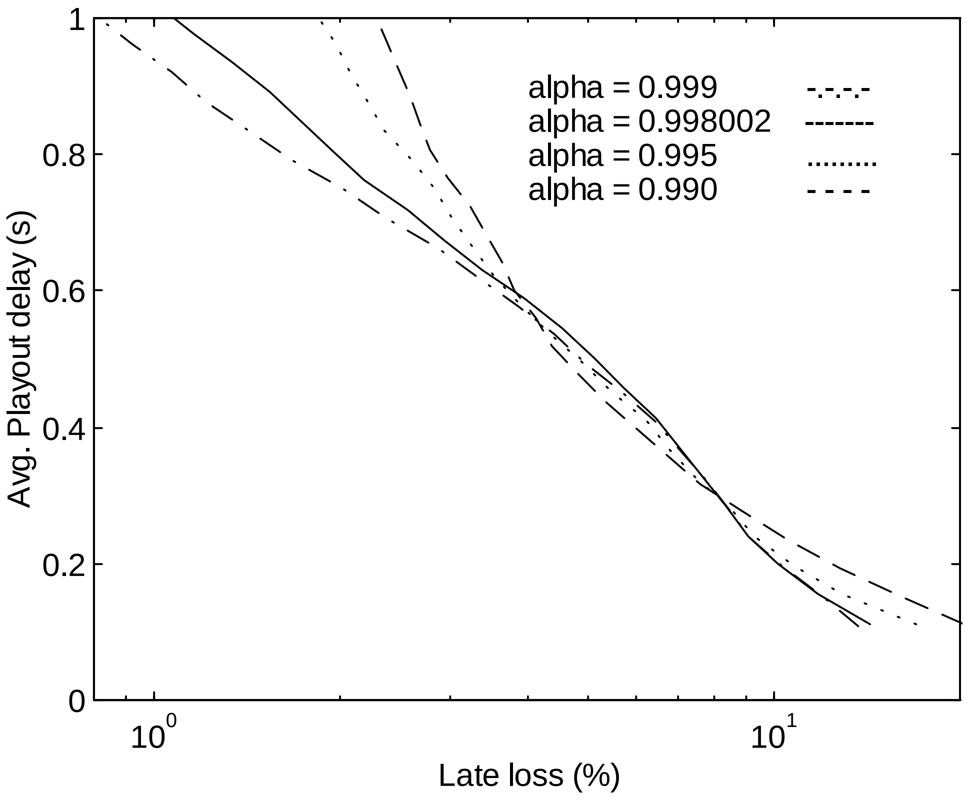


Figure 6-10: Algorithm 1 on trace 3

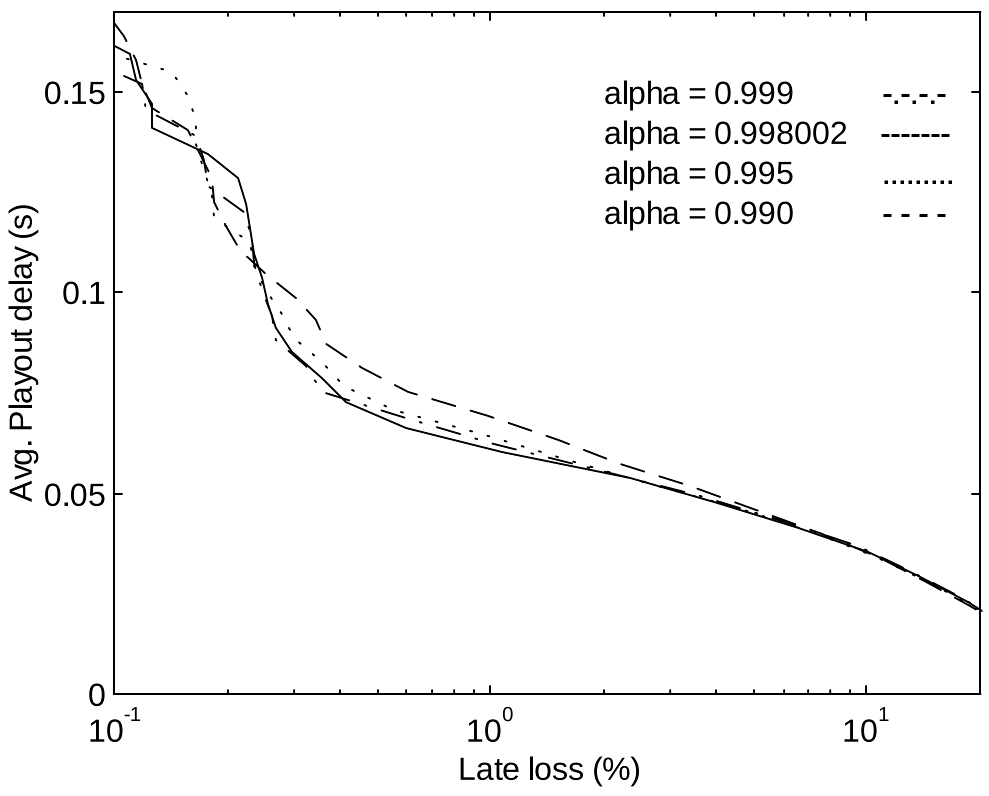


Figure 6-11: Algorithm 1 on trace 4

On traces 1 and 2, $\alpha = 0,999$ gave the best performance. On trace 3, $\alpha = 0,990$ was best in the area of loss rate around 5 %, which would be the optimal operating point in this case, and on trace 4, best performance was achieved with $\alpha = 0,998002$. With all other traces except with 3, performance degrades when α is reduced under $\alpha = 0,998002$.

6.3.3. Performance of algorithm 2

Here β is also varied from 0,5 to 20 at the steps of 0,5. In [28], α is suggested to be 0,875. Here we compare α with values 0,875, 0,900, 0,950 and 0,990. Figures 6-12 to 6-15 show the results. Matlab code for algorithm 2 is presented in Appendix 7.

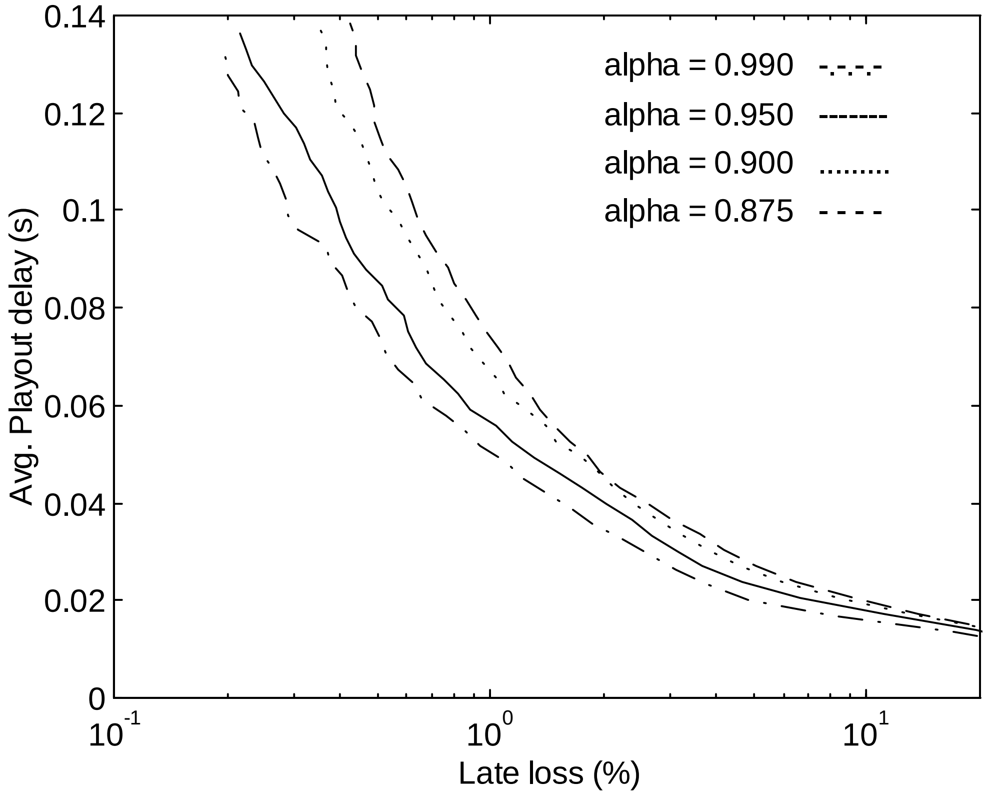


Figure 6-12: Algorithm 2 on trace 1

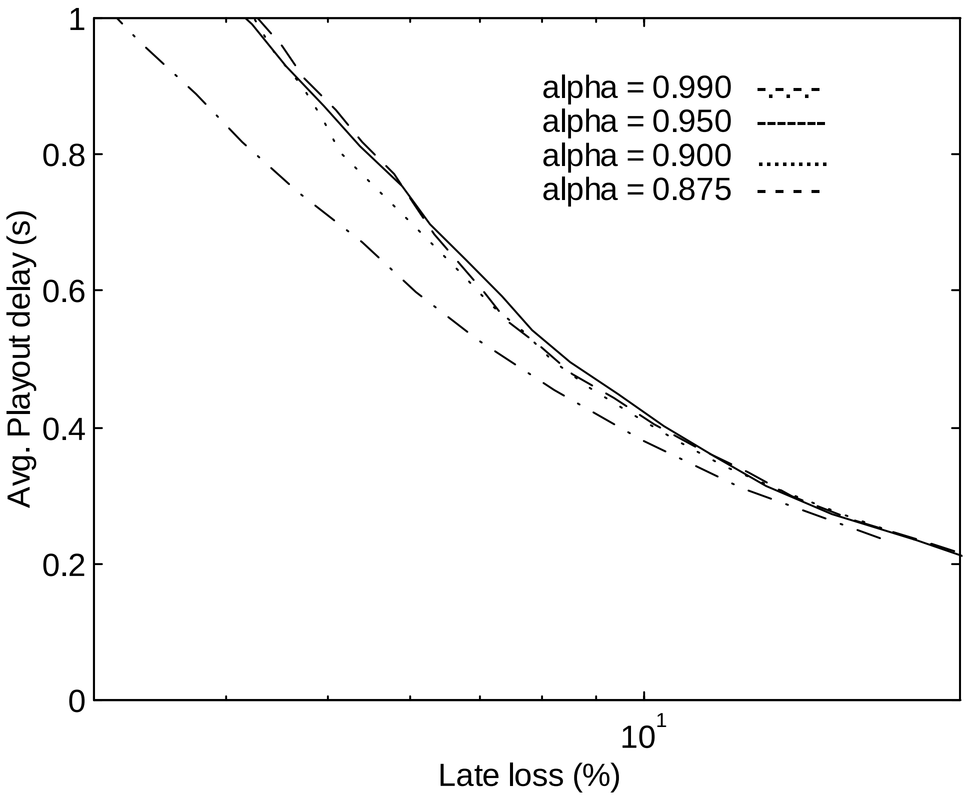


Figure 6-13: Algorithm 2 on trace 2

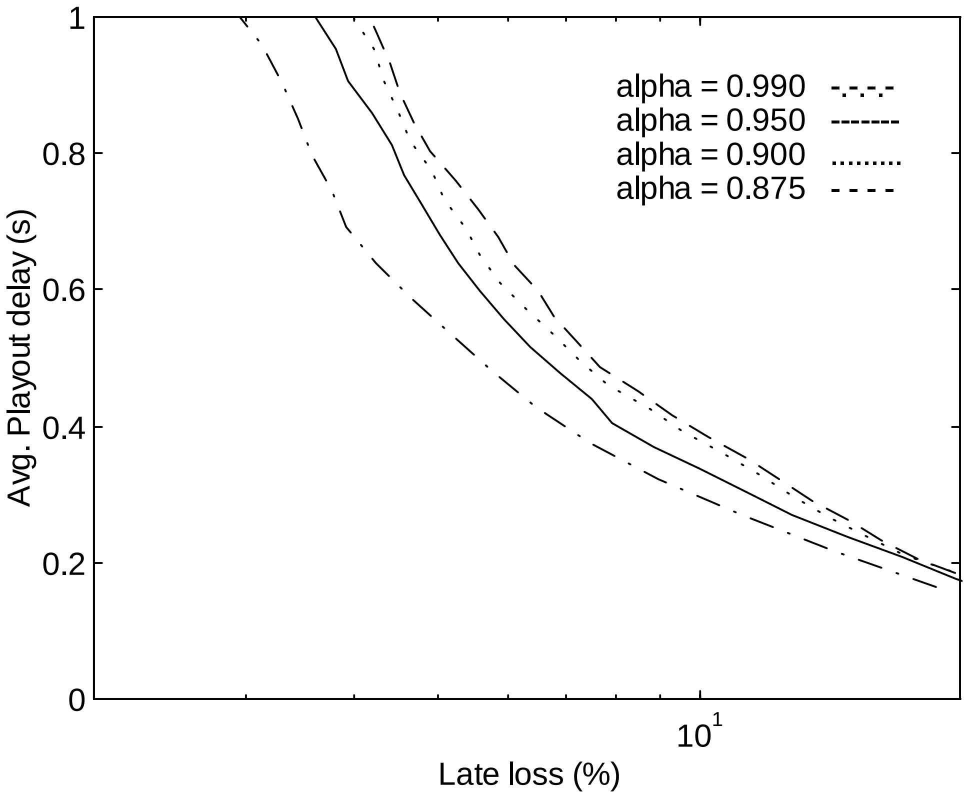


Figure 6-14: Algorithm 2 on trace 3

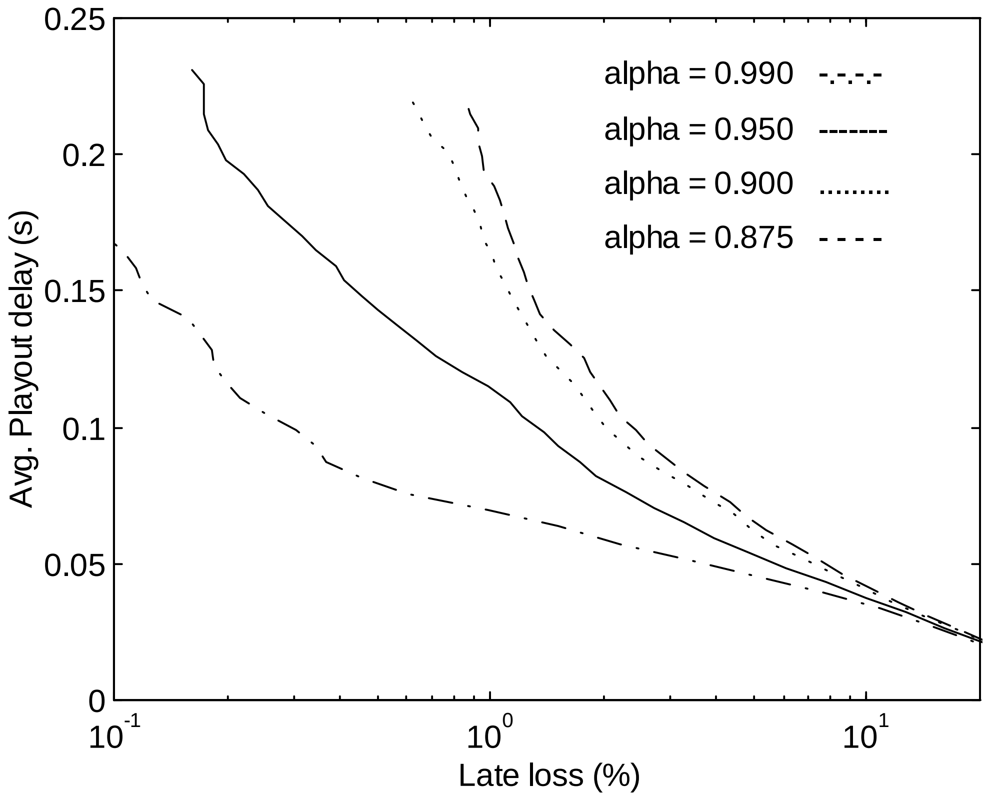


Figure 6-15: Algorithm 2 on trace 4

Because the actual difference between algorithms 1 and 2 is that algorithm 2 provides an additional operating mode for adapting to spikes, it is worth examining how well this spike detection scheme works with our traces. Table 6-10 presents the amount of detected spikes with each trace and each value of α.

Table 6-10: Numbers of detected spikes on algorithm 2

	α = 0,875	α = 0,900	α = 0,950	α = 0,990
Trace 1	0	0	0	0
Trace 2	31	31	31	20
Trace 3	13	13	11	8
Trace 4	0	0	0	0

With traces 1 and 4, the algorithm doesn't detect any spikes because the spike-detection threshold is so high compared to the delay variance of these traces. In these cases algorithm 2 thus reduces to algorithm 1 with different values of α. With traces 2 and 3, the amount of detected spikes varies between 8 and 31. When α is increased, it slows down the convergence of the playout delay back to normal level after detection of a spike. However, the best performance on all traces is achieved with the largest value of α because most of the time the algorithm operates in normal mode.

6.3.4. Performance of algorithm 3

In algorithm 3, the trade-off between average playout delay and loss rate is controlled by the percentile point *q*. Values used for q were: 1, 0,9995, 0,9990, 0,998, 0,995, 0,990, 0,98, 0,97, 0,96, 0,94, 0,92, 0,90, 0,88, 0,86, 0,83 and 0,80. The length of the delay distribution window, parameter *w*, is given values of : 500, 1000, 2000 and 5000. The resolution of the delay distribution was set to 10 ms.

As discussed in Section 4.3.3., algorithm 3 needs some time to first collect the delay statistics of the received packets. These results are obtained by calculating the playout delays for packets from number *w*+1 onwards. Matlab code for algorithm 3 is presented in Appendix 8. Results are shown in Figures 6-16 to 6-19.

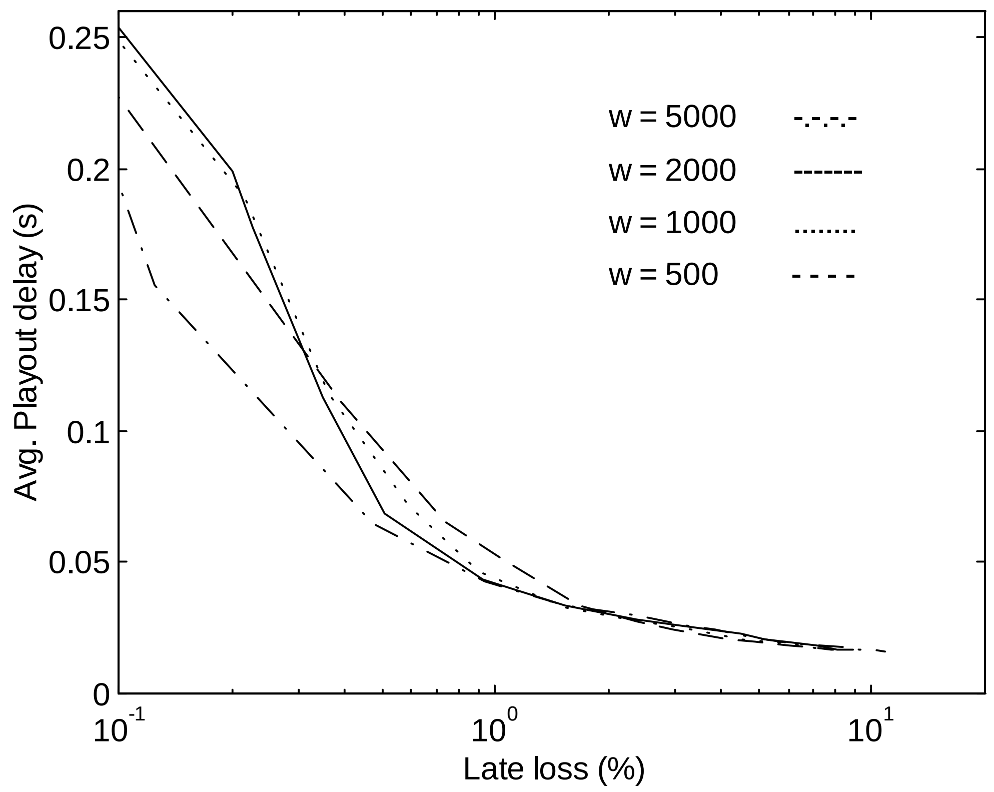


Figure 6-16: Algorithm 3 on trace 1

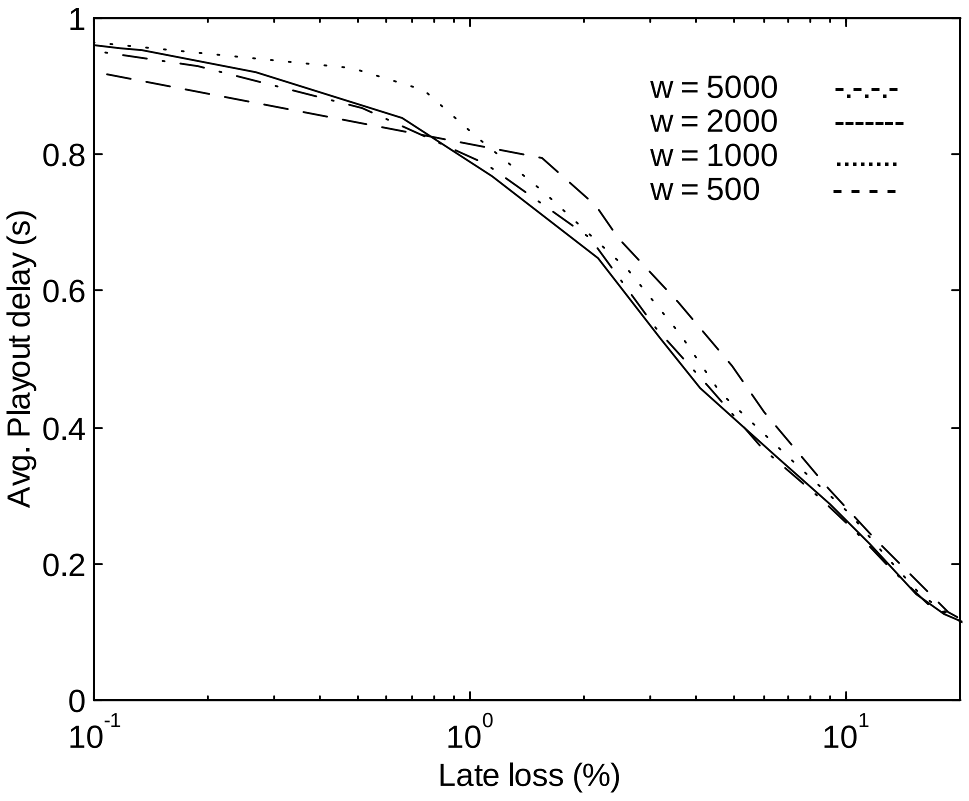


Figure 6-17: Algorithm 3 on trace 2

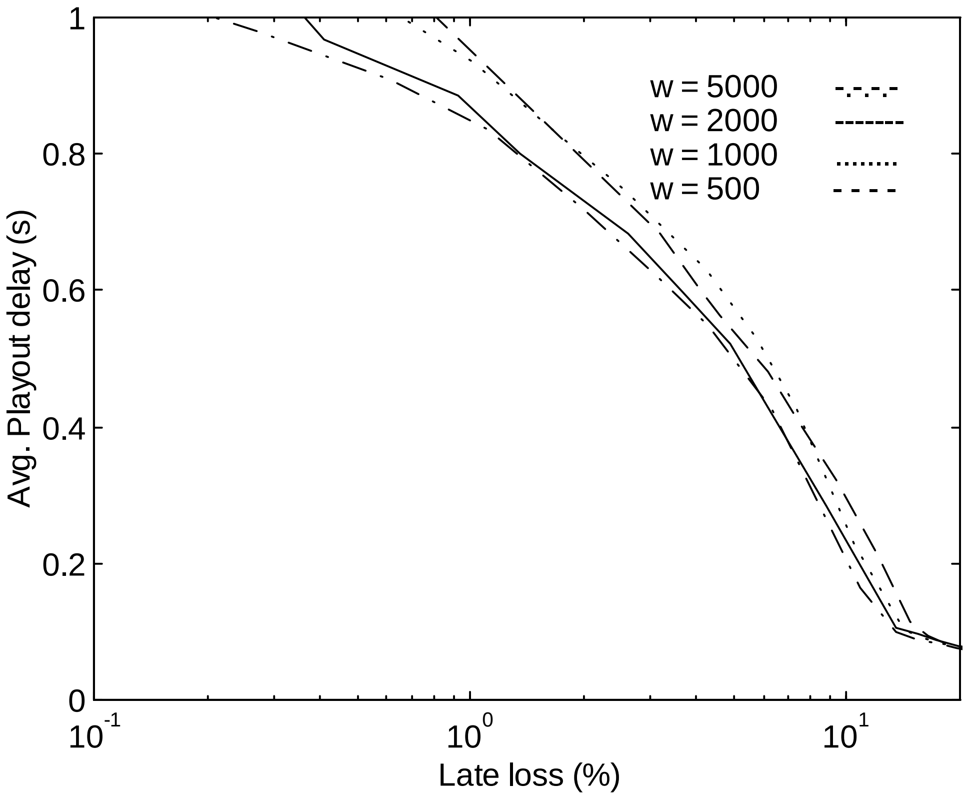


Figure 6-18: Algorithm 3 on trace 3

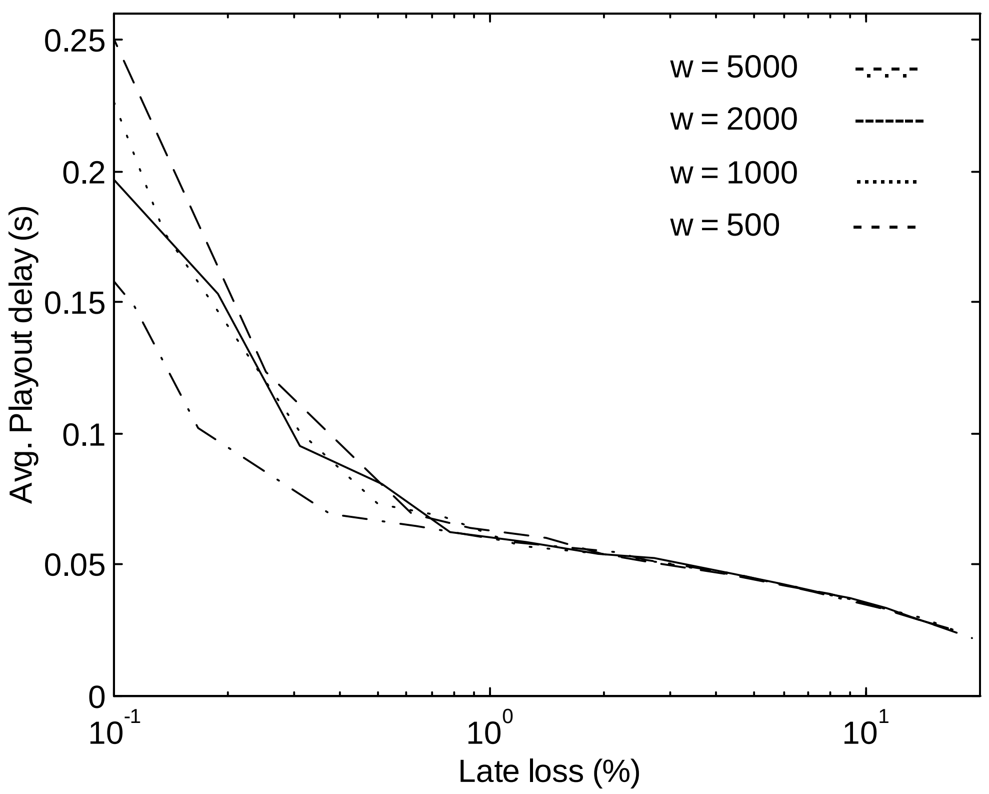


Figure 6-19: Algorithm 3 on trace 4

Performance of algorithm 3 improves as *w* increases. Values of *w* > 5000 didn't seem to improve the performance any further in our experiments. With *w* < 1000, even small decrementations of *q* under 1 tend to give a large increase in the loss ratio. This can be seen for example in Fig. 6-17 where it looks like curve with *w* = 500 would give the best performance on small loss rates, but in fact the curve just takes a short-cut because already the value of *q* = 0,9995 gives over 1 per cent loss rate.

It is also useful to examine how the spike detection works in this algorithm. Tables 6-11 to 6-14 present the amount of detected spikes on different traces and parameter values.

Table 6-11: Numbers of detected spikes on algorithm 3 and trace 1

	W = 500	*W = 1000*	*W = 2000*	*W = 5000*
Q = 1,0000	0	0	0	0
Q = 0,9995	7	2	3	0
Q = 0,9990	7	2	3	1
Q = 0,998	7	6	5	2
Q = 0,995	12	8	11	9
Q = 0,990	15	17	19	18
Q = 0,98	22	22	23	19
Q = 0,97	33	29	30	26
Q = 0,96	39	41	33	29
Q = 0,94	46	46	44	32
Q = 0,92	47	47	47	38
Q = 0,90	49	50	49	38
Q = 0,88	50	51	52	44
Q = 0,86	53	52	55	45
Q = 0,83	55	54	56	47
Q = 0,80	57	56	56	51

Table 6-12: Numbers of detected spikes on algorithm 3 and trace 2

	W = 500	*W = 1000*	*W = 2000*	*W = 5000*
Q = 1,0000	0	0	0	0
Q = 0,9995	2	0	0	0
Q = 0,9990	2	0	0	0
Q = 0,998	2	0	0	0
Q = 0,995	3	0	0	0
Q = 0,990	3	0	0	0
Q = 0,98	4	0	0	0
Q = 0,97	9	0	0	0
Q = 0,96	11	2	1	0
Q = 0,94	27	16	8	0
Q = 0,92	49	33	21	16
Q = 0,90	68	54	36	26
Q = 0,88	103	98	82	45
Q = 0,86	136	124	116	76
Q = 0,83	168	157	155	119
Q = 0,80	199	177	171	148

Table 6-13: Numbers of detected spikes on algorithm 3 and trace 3

	W = 500	*W = 1000*	*W = 2000*	*W = 5000*
Q = 1,0000	0	0	0	0
Q = 0,9995	7	0	0	0
Q = 0,9990	7	0	0	0
Q = 0,998	7	0	0	0
Q = 0,995	12	2	0	0
Q = 0,990	14	2	0	0
Q = 0,98	19	6	0	0
Q = 0,97	33	11	0	0
Q = 0,96	47	21	0	0
Q = 0,94	77	66	33	0
Q = 0,92	101	99	98	51
Q = 0,90	107	106	97	82
Q = 0,88	112	112	100	88
Q = 0,86	114	112	105	91
Q = 0,83	126	126	111	92
Q = 0,80	167	145	117	98

Table 6-14: Numbers of detected spikes on algorithm 3 and trace 4

	W = 500	*W = 1000*	*W = 2000*	*W = 5000*
Q = 1,0000	0	0	0	0
Q = 0,9995	2	2	1	0
Q = 0,9990	2	2	1	0
Q = 0,998	2	5	4	4
Q = 0,995	8	9	8	7
Q = 0,990	9	10	10	8
Q = 0,98	10	11	10	8
Q = 0,97	11	11	11	11
Q = 0,96	11	13	14	11
Q = 0,94	13	16	17	13
Q = 0,92	17	18	17	13
Q = 0,90	21	19	20	15
Q = 0,88	22	23	23	17
Q = 0,86	23	25	25	19
Q = 0,83	109	26	27	21
Q = 0,80	463	216	30	22

It can be noted that the number of detected spikes increases quite fast as q decreases. Also the number of detected spikes decreases as w increases. This is obvious because with a larger w and q, the variance of the playout delay stays smaller.

If we compare the spike detection techniques used in algorithms 2 and 3, it can be noted that in algorithm 2 the threshold is almost fixed and relatively high because of the used absolute time component. In algorithm 3 the beginning of a spike is detected by a large enough change in delay compared to previous playout delay and therefore it adapts the threshold according to the current delay distribution.

Also in algorithm 2 it takes some time to converge back to the earlier delay level after a spike, whereas in algorithm 3 we return directly to the earlier situation. Thus algorithm 2 introduces unnecessary additional delay in this concept.

6.3.5. Comparison of algorithms 1-3

Here we compare on each trace algoritms 1-3 with parameters that gave the best performance in previous sections. For algorithm 1, α = 0,999 is used for traces 1 and 2, α = 0,990 is used for trace 3, and α = 0,998002 is used for trace 4. For algorithm 2, α = 0,990 in all traces and for algorithm 3, w = 5000 in all traces. Results are shown in Figures 6-20 to 6-23.

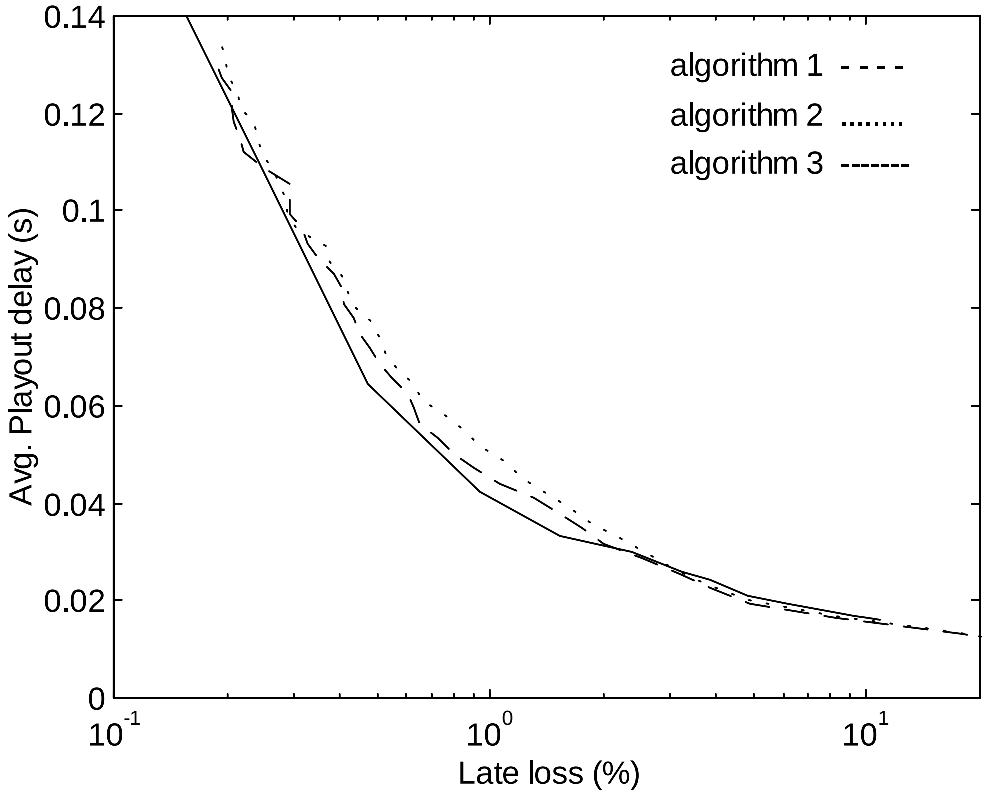


Figure 6-20: Algorithms 1-3 on trace 1

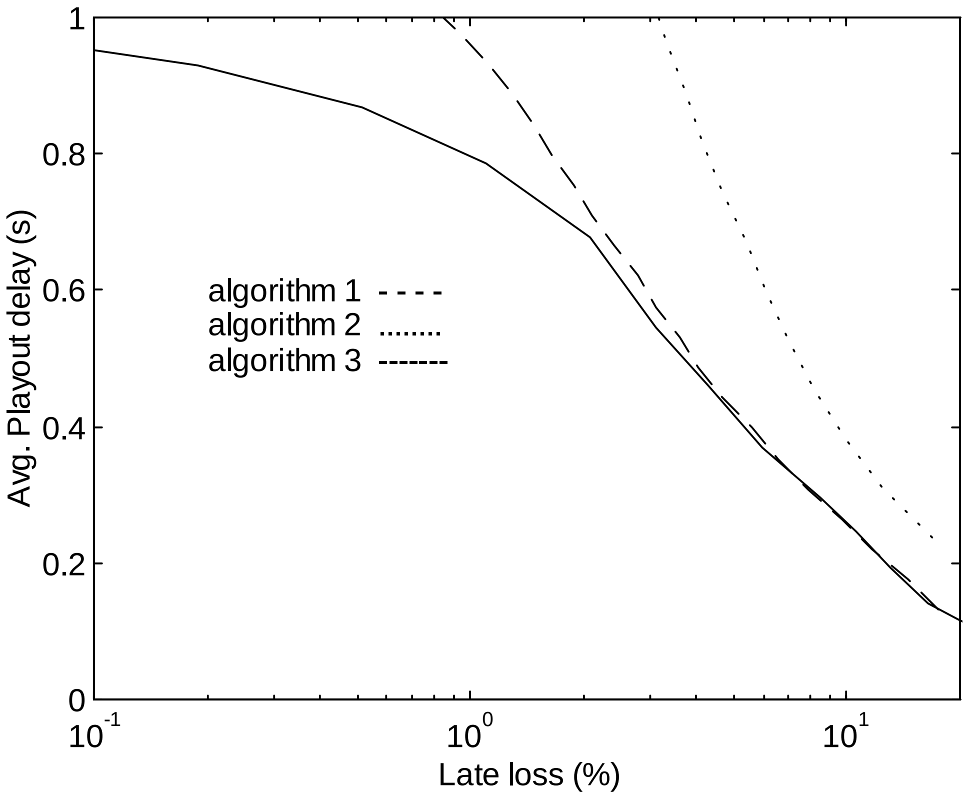


Figure 6-21: Algorithms 1-3 on trace 2

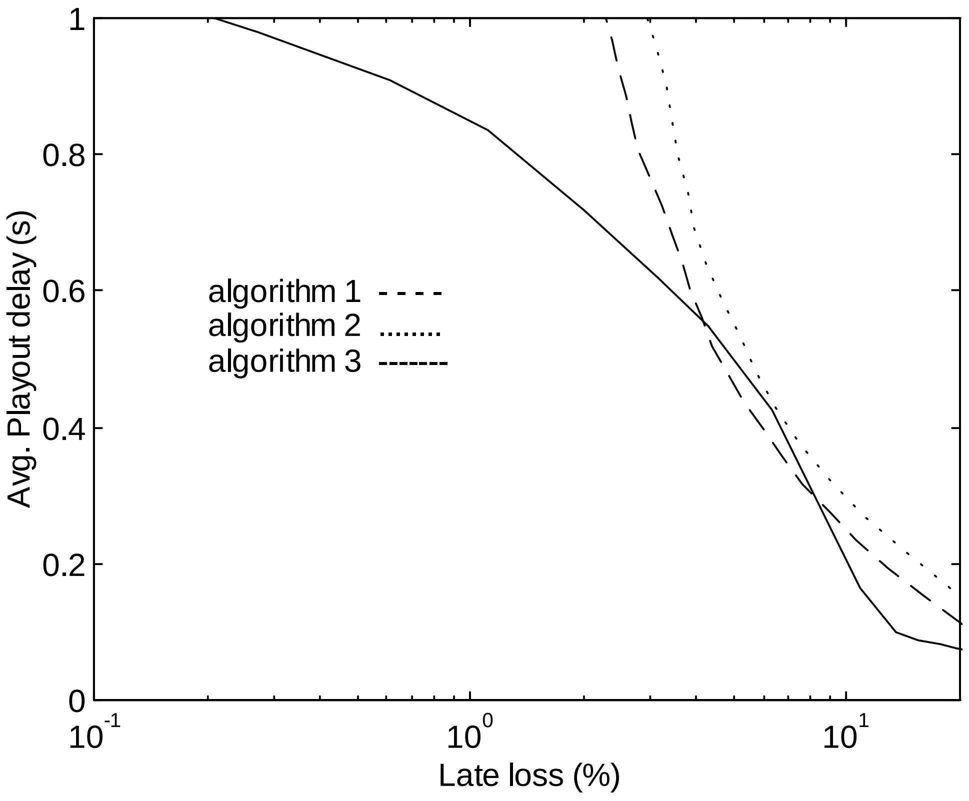


Figure 6-22: Algorithms 1-3 on trace 3

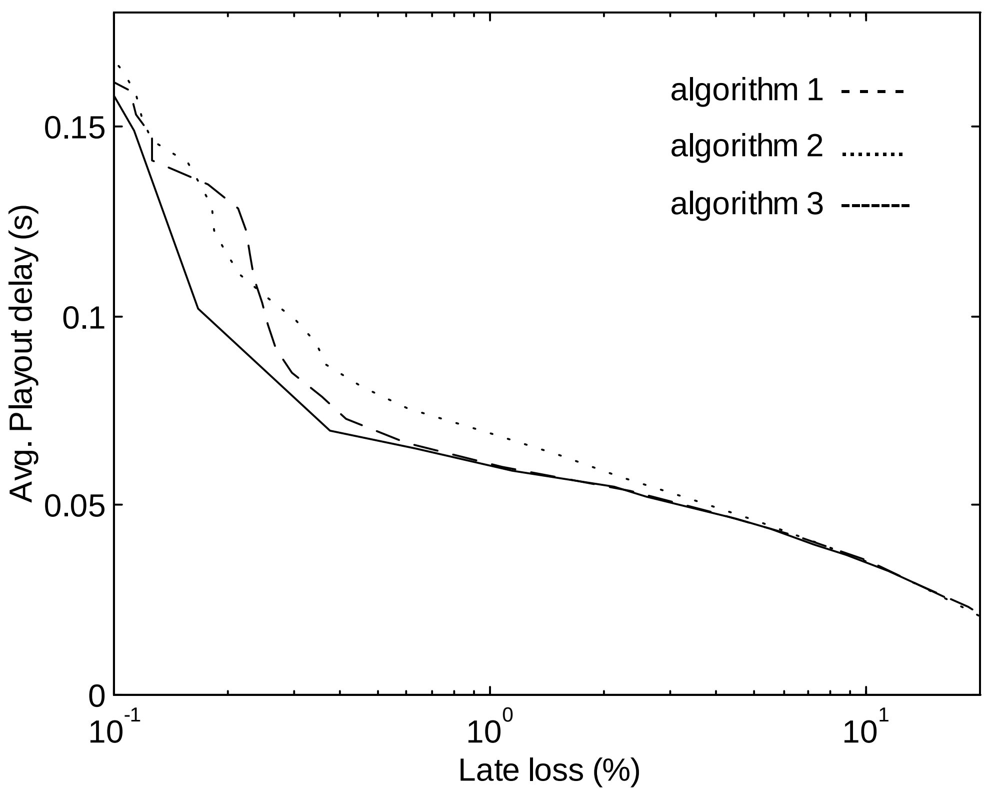


Figure 6-23: Algorithms 1-3 on trace 4

It can be seen that algorithm 3 gives the best performance over the other two algorithms on all traces except on trace 3, where algorithm 1 gives the best performance at the loss rates of 4-8 %. The differences between the performances of different algorithms seemed to increase with smaller loss rates and greater delay variance. With traces 2 and 3, algorithms 1 and 3 gave almost similar performance at loss ratios over 4 %, but when loss ratios decrease, algorithm 3 performs a lot better.

Algorithm 2 performes quite poorly. This is because the spike detection mode doesn’t work like desired with our traces. The spikes that we could generate in our experiments were not as long measured in packets as this algorithm is planned to be used with. The spikes don’t exceed very often to several talkspurts and thus the algorithm only seems to increase the delay significantly but doesn’t reduce the loss ratio so much. Also in normal mode algorithm 2 gives worse performance than algorithm 1 because of the smaller value of α.

Traces 2 and 3 tend to give very high playout delays because of the high network load. If we had an IP voice connection over this kind of a network, the parameters of the playout algorithms should be set so that the playout delay would be reduced at the expense of the packet loss. Packet losses up to 5 % can be tolerated. This would give playout delays around 400 ms for traces 2 and 3 . Conversation would still be possible but not very convenient because of the high delays and reduced sound quality.

6.3.6. Algorithm 4

With algorithm 3, the problem is that it first needs to collect some delay statistics. In a real-time implementation it is not possible to use algorithm 3 from the beginning of a call. Therefore we can combine together algorithms 1 and 3 and the resulting algorithm is called here algorithm 4. The idea is simply to calculate the playout delays with algorithm 1 until we have received 5000 voice packets and then switch to algorithm 3. The switch can’t be done earlier because then algorithm 3 would probably give worse results than algorithm 1. The extension to algorithm 1 is that we use the two operating modes from algorithm 3 since the beginning of the call. The Matlab code for algorithm 4 is presented in Appendix 9. In Figures 6-24 to 6-27 we present the performance of algorithm 4 compared with algorithms 1 and 3.

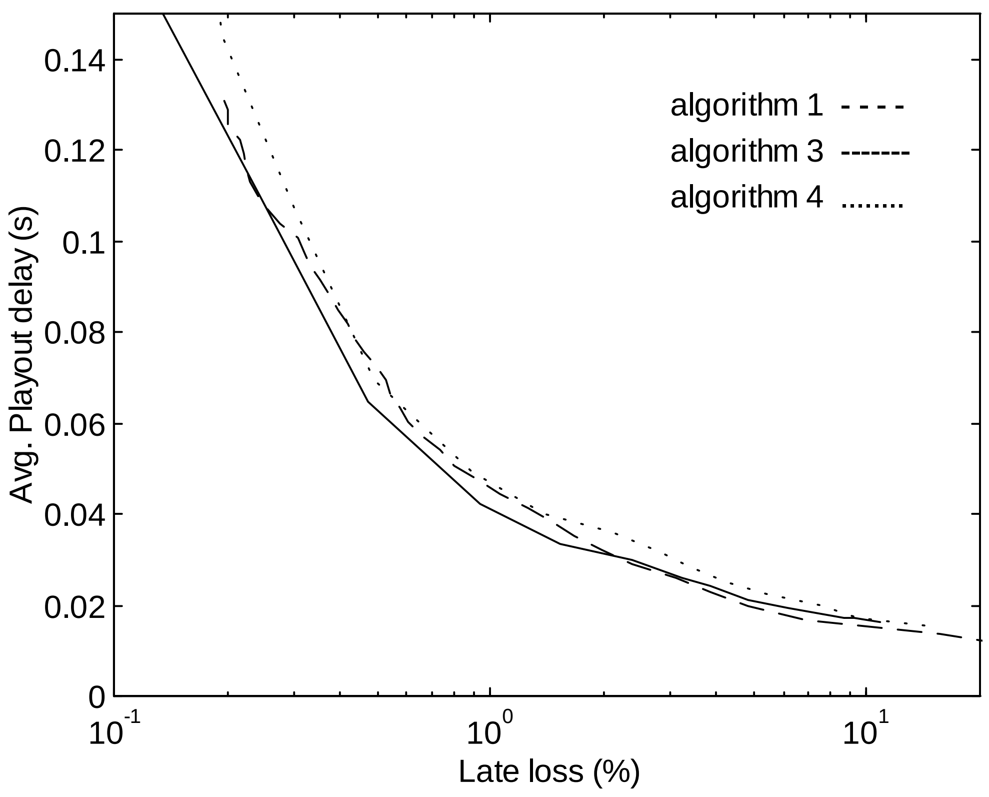


Figure 6-24: Algorithms 1, 3 and 4 on trace 1

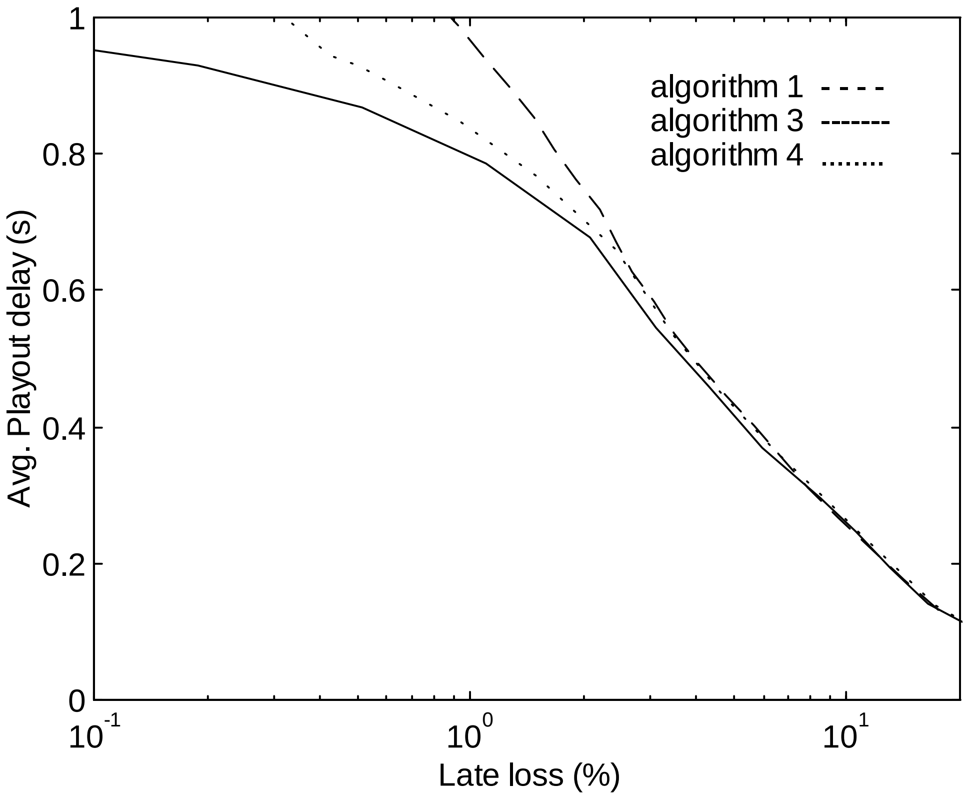


Figure 6-25: Algorithms 1, 3 and 4 on trace 2

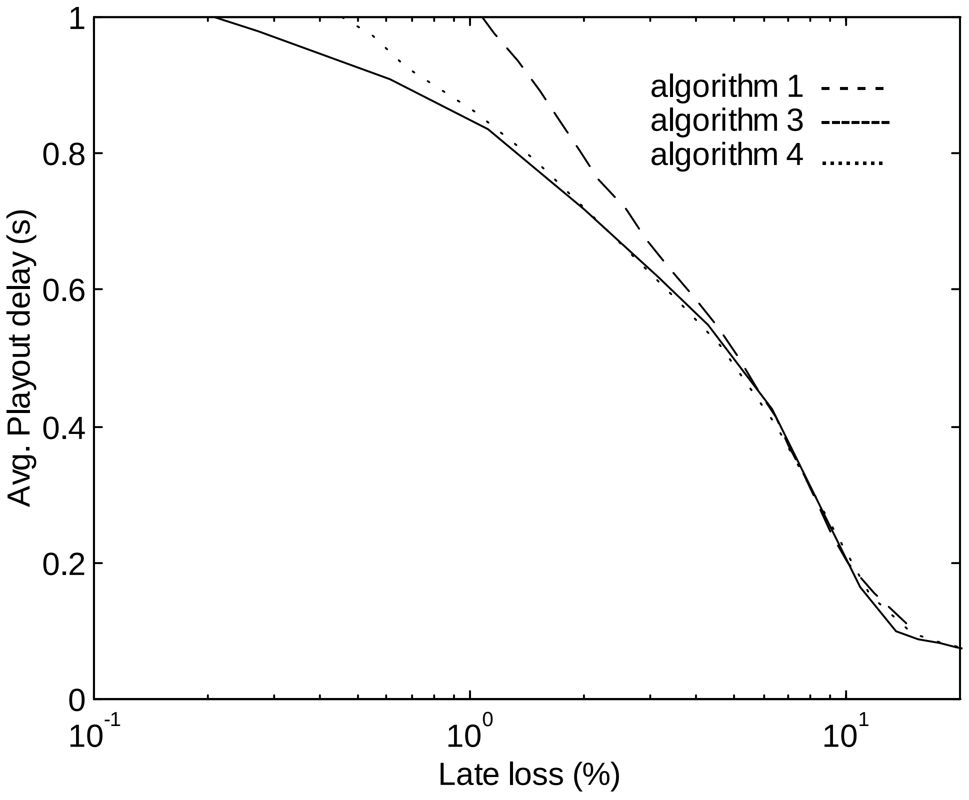


Figure 6-26: Algorithms 1, 3 and 4 on trace 3

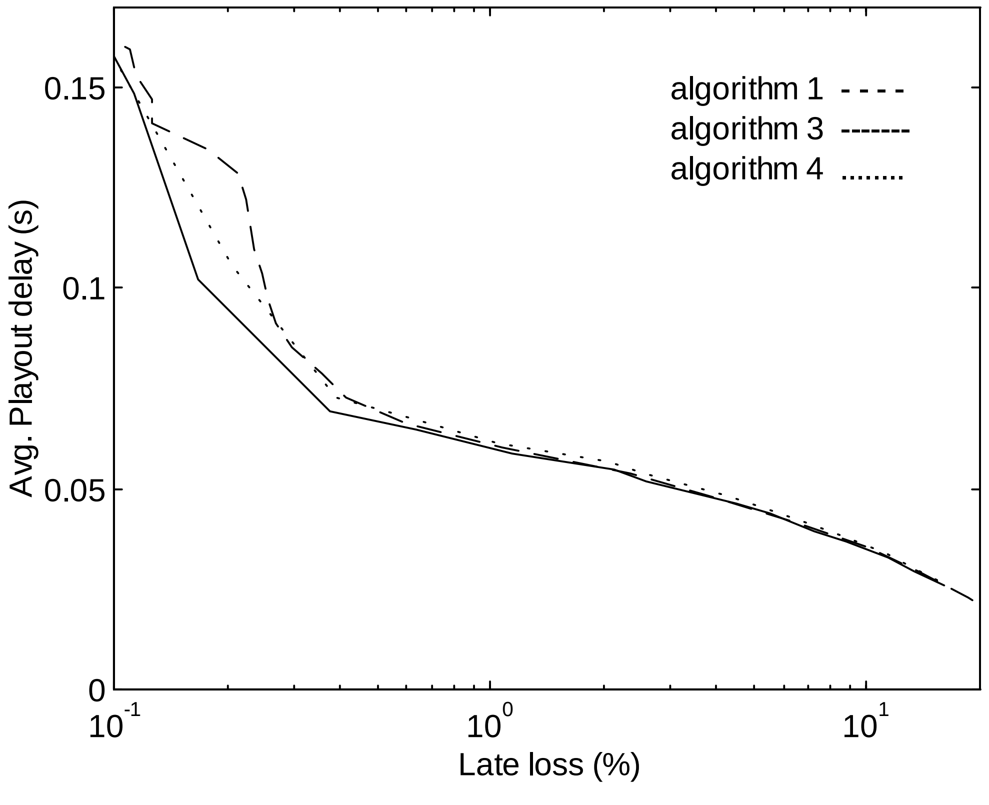


Figure 6-27: Algorithms 1, 3 and 4 on trace 4

Logically, algorithm 4 shows performance between algorithms 1 and 3 except with trace 1 where the results are about the same for algorithms 1 and 4. In the higher scale of packet loss, all three algorithms perform equally well. When packet loss reduces, algorithm 4's performance improves compared to algorithm 1.

If we are using 20 ms voice packets then 5000 packets would mean 100 seconds of voice. When we consider that in average 40 % of the time the sender is active, this means that the switch from algorithm 1 to algorithm 3 happens approximately 250 seconds after the beginning of a call. The actual benefit of algorithm 4 is obtained during long calls, with duration over 4 minutes. But compared to algorithms 1 and 2, the use of a better spike detection mode from algorithm 3, improves the presentation from the beginning of a call.

6.3.7. Summary

We compared the performance of three different playout algorithms that were found in literature. The used performance metric was average playout delay vs. packet playout loss. These algorithms were simulated on different network caused delay distributions. Delays were generated by loading Ethernet with Radcom protocol analyzer.

Algorithm 1 is based on stochastic gradient algorithm and is basically a linear filter. Algorithm 2 is actually algorithm 1 extended with a second operational mode to detect rapid changes in network delay characteristics. Algorithm 3 is based on calculating playout delays based on previous delay history.

In our simulations, algorithm 2 gave the worst performance. In [28] algorithm 2 was shown to outperform algorithm 1 with most of the used traces. The reason is that in [28] real Internet traces were used and in our experiments the network delays were simulated in Ethernet. In our measurements it was not possible to create "real" spikes that would have spread into several talkspurts and thus take advantage of algorithm 2's properties.

In our simulations, algorithm 3 gave the best results compared with algorithms 1 and 2. Similar results were reported in [29]. The problem with algorithm 3 is that it is not real-time implementable as such. Therefore we presented a new algorithm which combines together algorithms 1 and 3. This algorithm is shown to perform better that the other two real-time algorithms. Most advantage of this algorithm is got when the duration of a call exceeds 250 seconds. It's not uncommon for a phone call to last for example 30 minutes and in such case we will get significant benefit by using this algorithm.

With playout algorithms, it is essential that the parameters that control the playot delay/ late loss - ratio are configured appropriately. In Nevot 3.35 playout algorithm 1 is used. User can define β as an integer between 1 and 10. The default value is 4. So it is up to the user to reconfigure β to match with the changing network delay characteristics. As an improvement to the existing playout algorithms, an algorithm could keep statistics of the average playout delay and late loss rate and adapt its parameters to give the best possible performance with current network delay characteristics.

7. CONCLUSIONS AND FUTURE WORK

In this thesis we concentrated on the delays in an IP voice terminal. We measured the processing delays in a Voice over IP application software using different audio codecs. Even with the heaviest codec, the contribution of processing delay to the end-to-end delay stayed under 4 ms.

We also studied the end-to-end delay between two Sun Ultra workstations in a situation where the network between the workstations was unloaded and practically all delay was generated in the workstations.

The components of the end-to-end delay were measured and a significant part of the delay was found out to be generated in the receiving workstation where the application software starts collecting voice packets to the playout buffer in order to avoid buffer underflows. Other components of the end-to-end delay were framing delay, processing delay, other buffering delays, HW delay in the soundcard and the delay caused by operating system if VoIP software was run as a time-sharing class process.

In our measurements, the difference in the end-to-end delays between real-time and time-sharing class processes was relatively small, between 0 and 3 ms, but these measurements were performed in the summertime when the workstations were lightly loaded.

Our experiments show that the Sun Ultra platform provides an environment where it is possible to provide bounds on the delays presented in the workstations. This can be accomplished with real-time scheduled processes that are provided by the operating system. If the application software is implemented so that it presents no additional delays and the used network connection is lightly loaded, it is possible to achieve end-to-end delays in the order of 30-40 milliseconds using 20 ms packet size.

We also made comparison of different playout algorithms under different network delay characteristics. The best performance in our simulations was obtained by an algorithm that was based on previous delay history. This algorithm was not real-time implementable as such and therefore we presented a new algorithm that was a combination of this and an existing real-time algorithm. Our algorithm was shown to outperform the other existing real-time algorithms that were compared in our studies. This algorithm gives best performance with calls that durate over 250 seconds. During the first 250 seconds, delay statistics are collected and from there on, playout delays are calculated using delay distribution of previous 5000 packets.

To further improve the performance of our playout algorithm, it could keep track of the average playout delay and late loss rate and adapt its parameters according to the network delay characteristics. Future work also includes studying delays in PC evironment with Linux operating system.

REFERENCES

[1] Yletyinen, T., “Quality of voice over IP”, Master’s thesis, Helsinki University of technology, Telecomm tech., 1997.

[2] Gold, B., “Digital Speech Networks”, IEEE Proceedings, vol. 65, no. 12, Dec. 1977

[3] Schulzrinne, H., “Voice communication across the Internet: A network voice terminal”, Univ. of Massachusetts, USA, 1992.

[4] Stallings, W., “Data and computer communications, 4th edition”, Macmillan Publishing Company, New York, USA, 1994.

[5] Deering, S., Hinden, R., Internet Protocol, Version 6 (IPv6) Specification, RFC 1883, Dec. 1995.

[6] Deering, S., Hinden, R., Internet Protocol, Version 6 (IPv6) Specification, Internet Draft, draft-ietf-ipngwg-ipng-spec-v2-00.txt , July 1997.

[7] Shulzrinne, H., Casner., S., Frederick, R., Jacobsen, V., “A Transport Protocol for Realtime Applications”, Network Working Group Request for comments:1889, Internet Engineering Task Force , Jan. 1996.

[8] Thom, G., “H.323: The Multimedia Communication Standard for Local Area Networks”, IEEE Communications, Dec. 1996.

[9] Schulzrinne, H., “Nevot Implementation and program structure”, GMD Fokus, Berlin, 1996.

[10] Cox, Richard V., “Three new speech coders from the ITU cover a range of applications”, IEEE Communications, Vol.35, No. 9, pp. 40-47, Sep. 1997.

[11] Perkins, M., Evans, K., Pascal, D., Thorpe, L., “Characterizing the Subjective Performance of the ITU-T 8 kb/s Speech Coding Algorithm - ITU-T G.729”, IEEE Communications, Vol.35, No. 9 , pp. 74-81, Sep. 1997.

[12] ITU-T Recommendation P.830, “Subjective performance assessment of telephone-band and wideband digital codecs”, 1996

[13] ITU-T Recommendation P.800, “Methods for subjective determination of transmission quality”, 1996

[14] Flanagan, J.L. et al., “Speech coding”, IEEE Trans.Comm., Vol 27, No. 4, April 1979, pp. 710-737.

[15] Rabiner, L.R., Schafer, R.W., “Digital processing of speech signals”, Prentice-Hall, 1978.

[16] Yletyinen, T., “IP- puhe. Katsaus standardointiin, ohjelmistoihin ja laitteistoihin”, Lab of Telecommunications Technology, Helsinki University of Technology, 1997.
[17] Couch, L., “Digital and analog communication systems”, Prentice-Hall, USA, 1997.
[18] ITU-T Recommendation G.711, “Pulse code modulation (PCM) of voice frequencies”, 1993.
[19] ITU-T Recommendation G.726, “40, 32, 24, 16 kbit/s Adaptive Differerential Pulse Code Modulation (ADPCM)”, 1990
[20] Mouly, M., Pautet, M-B.,”The GSM system for mobile communications”, France, 1992.
[21] ITU-T Recommendation G.723.1, “Dual rate speech coder for multimedia communications transmitting at 5,3 and 6,3 kbit/s”, 1996.
[22] ITU-T Recommendation G.729, “Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear-prediction (CS-ACELP).
[23] Telecommunication Standardization Sector Of ITU. ITU-T Recommendation G.114. Technical report, International Telecommunication Union, March 1993.
[24] Jayant, N., “Effects of packet loss on waveform coded speech”, Fifth Int. Conference on Computer Communications, pp.275-280, Atlanta, GA, October 1980.
[25] Montgomery, W., “Techiques for packet voice synchronization”, IEEE Journal on Selected Areas In Communications, 6(1), pp.1022-1028, December 1983.
[26] Alvarez-Cuevas, F., Bertran, M., Oller, F., Selga, J., “Voice synchronization in packet switching networks”, IEEE Networks Magazine, 7(5), pp.20-25, September 1993.
[27] Bolot, J., “End-to-end packet delay and loss behavior in the Internet”, Proceedings of ACM SIGCOMM ’93, pp.289-298, San Francisco, CA, September 1993.
[28] Ramjee, R., Kurose, J., Towsley, D., Schulzrinne, H., “Adaptive playout mechanisms for packetized audio applications in wide area networks”, Proc. IEEE Infocom ’94, Montreal, Canada, April 1994.
[29] Moon, S., Kurose, J., Towsley, D., “Packet Audio Playout Delay Adjustment: Performance Bounds and Algorithms”, Technical Paper, Dept. of Computer Science, Univ. of Massachusetts at Amherst, 1995.
[30] Ljung, L., Söderström, T., “Theory and practice of recursive identification”, MIT press, 1983.
[31] Tanenbaum, A., “Modern operating systems”, Prentice-Hall, New Jersey, USA, 1992
[32] “Sun Solaris 2.4 System services guide”, Sun Microsystems, California, USA, 1994.

[33] http://www.spec.org/osg/cpu95, October 1998.

APPENDIX 1:

Profile file of Nevot CPU consumption with PCM audio coding

```
%Time Seconds Cumsecs  #Calls msec/call Name

 49.8   1.58   1.58  30541    0.0517  audio_stats
  9.5   0.30   1.88  15252    0.0197  mix2_pcmu
  5.7   0.18   2.06                   _mcount
  3.8   0.12   2.18                   Tk_DoOneEvent
  2.8   0.09   2.27  30485    0.0030  ring_mix
  2.5   0.08   2.35  15355    0.0052  rx
  2.5   0.08   2.43 131328    0.0006  pcmu_to_l14
  1.9   0.06   2.49  15292    0.0039  tx_local_audio
  1.3   0.04   2.53  15258    0.0026  tx_packet_session
  1.3   0.04   2.57  15190    0.0026  tx_packet
  1.3   0.04   2.61  30485    0.0013  bytes2samples
  0.9   0.03   2.64                   Tcl_ConvertElement
  0.9   0.03   2.67  15190    0.0020  agc
  0.9   0.03   2.70                   TclParseWords
  0.9   0.03   2.73  30583    0.0010  ring_action
  0.9   0.03   2.76  15190    0.0020  rtp_write_data
  0.6   0.02   2.78                   ValueToPixel
  0.6   0.02   2.80  15258    0.0013  tx_member
  0.6   0.02   2.82  28039    0.0007  tx
  0.6   0.02   2.84  30482    0.0007  peakmeter
  0.6   0.02   2.86  15292    0.0013  play_local
  0.6   0.02   2.88                   gcc2_compiled., strtod
  0.6   0.02   2.90  15295    0.0013  rx_stats
  0.6   0.02   2.92  15417    0.0013  member_find
  0.6   0.02   2.94   6164    0.003   event
  0.3   0.01   2.95                   $2
  0.3   0.01   2.96                   Tk_BindEvent
  0.3   0.01   2.97                   Tcl_ScanElement
  0.3   0.01   2.98                   DisplayVerticalMeter
  0.3   0.01   2.99  15355    0.0007  UDP_read
  0.3   0.01   3.00  43394    0.0002  file_handler
  0.3   0.01   3.01                   DisplayButton
  0.3   0.01   3.02  30483    0.0003  ring_cpy
  0.3   0.01   3.03                   Tk_Preserve, gcc2_compiled.
  0.3   0.01   3.04  58961    0.0002  debug
  0.3   0.01   3.05      1   10.      audio_stats_init
  0.3   0.01   3.06  15190    0.0007  tx_session_attach
  0.3   0.01   3.07  15258    0.0007  UDP_write
  0.3   0.01   3.08  31194    0.0003  htonl
  0.3   0.01   3.09                   Tcl_GetDouble
  0.3   0.01   3.10      1   10.      pcmu_linear_init
  0.3   0.01   3.11  30552    0.0003  m_get
  0.3   0.01   3.12  15354    0.0007  audio_silence
```

```
0.3   0.01   3.13                  TclParseBraces
0.3   0.01   3.14   15295  0.0007  rx_debug
0.3   0.01   3.15       1 10.      mix_init
0.3   0.01   3.16   15266  0.0007  tsap_ismulticast
0.3   0.01   3.17                  SetMeterValue
0.0   0.00   3.17      17  0.0     cmd_member
0.0   0.00   3.17      60  0.0     rtp_read_sdes
0.0   0.00   3.17      60  0.0     rtp_read_traffic
0.0   0.00   3.17      60  0.0     rtp_read_control
0.0   0.00   3.17   15295  0.0000  rtp_read_data
0.0   0.00   3.17      67  0.0     rtp_write_traffic
0.0   0.00   3.17      68  0.0     rtp_write_control
0.0   0.00   3.17       3  0.      rtp_change_audio
0.0   0.00   3.17       3  0.      rtp_init
0.0   0.00   3.17       2  0.      Audio_MemberListen
0.0   0.00   3.17      67  0.0     rtp_write_sdes
0.0   0.00   3.17   30485  0.0000  htons
0.0   0.00   3.17     603  0.00    rtcp_text
0.0   0.00   3.17      66  0.0     rtp_packets_exp
0.0   0.00   3.17   15415  0.0000  ntohl
0.0   0.00   3.17      67  0.0     rx_periodic
0.0   0.00   3.17       1  0.      rx_init
0.0   0.00   3.17       2  0.      cmd_simple
0.0   0.00   3.17   15295  0.0000  ntohs
0.0   0.00   3.17       1  0.      Audio_SessionFind
0.0   0.00   3.17       2  0.      talkspurt
0.0   0.00   3.17       1  0.      Audio_SessionDelete
0.0   0.00   3.17       1  0.      Audio_SessionAlloc
0.0   0.00   3.17       1  0.      Audio_SessionOpen
0.0   0.00   3.17       8  0.      Audio_SessionSetSDES
0.0   0.00   3.17       1  0.      Audio_SessionSetTTL
0.0   0.00   3.17       2  0.      Audio_MemberDelete
0.0   0.00   3.17       1  0.      cmd_session_audio
0.0   0.00   3.17       2  0.      Audio_SessionTalk
0.0   0.00   3.17       2  0.      Audio_MemberClose
0.0   0.00   3.17       2  0.      member_add
0.0   0.00   3.17      61  0.0     member_announce
0.0   0.00   3.17       1  0.      session_add_host
0.0   0.00   3.17       1  0.      Audio_exit
0.0   0.00   3.17       8  0.      rx_trace
0.0   0.00   3.17       1  0.      rx_config
0.0   0.00   3.17       4  0.      tx_trace
0.0   0.00   3.17   15351  0.0000  play_silence
0.0   0.00   3.17      26  0.0     cmd_session
0.0   0.00   3.17      67  0.0     tx_periodic_audio
0.0   0.00   3.17   15261  0.0000  session_ismulticast
0.0   0.00   3.17       1  0.      tx_start
0.0   0.00   3.17      65  0.0     tx_samples
0.0   0.00   3.17       1  0.      tx_init
0.0   0.00   3.17       1  0.      tx_packet_init
```

```
0.0   0.00   3.17       1   0.      tx_packet_close
0.0   0.00   3.17       1   0.      session_open
0.0   0.00   3.17       4   0.      cmd_general
0.0   0.00   3.17   30485   0.0000  talking_set
0.0   0.00   3.17       1   0.      Audio_SessionSetKey
0.0   0.00   3.17       1   0.      vat_init
0.0   0.00   3.17       3   0.      audio_change
0.0   0.00   3.17       1   0.      audio_out
0.0   0.00   3.17   30488   0.0000  audio_open
0.0   0.00   3.17       1   0.      Audio_SessionSetDuplex
0.0   0.00   3.17       1   0.      audio_close_idle
0.0   0.00   3.17       2   0.      Audio_SessionSetAudio
0.0   0.00   3.17       1   0.      Audio_SessionListen
0.0   0.00   3.17       6   0.      audio_chunk
0.0   0.00   3.17       1   0.      Tcl_AppInit
0.0   0.00   3.17       1   0.      ring_create
0.0   0.00   3.17    6137   0.000   vu_show
0.0   0.00   3.17       2   0.      talking_delete
0.0   0.00   3.17   15292   0.0000  talking_check
0.0   0.00   3.17       2   0.      ring_clear
0.0   0.00   3.17       1   0.      main
0.0   0.00   3.17     430   0.00    member_sdes
0.0   0.00   3.17       2   0.      talking_clear
0.0   0.00   3.17      66   0.0     timer_handler
0.0   0.00   3.17      68   0.0     notify_set_periodic_func
0.0   0.00   3.17       1   0.      tx_config
0.0   0.00   3.17       1   0.      cmd_init
0.0   0.00   3.17      67   0.0     tx_periodic
0.0   0.00   3.17       6   0.      agc_trace
0.0   0.00   3.17       1   0.      agc_init
0.0   0.00   3.17     130   0.00    audio_cmp
0.0   0.00   3.17     165   0.00    audio_descr
0.0   0.00   3.17       2   0.      audio_seconds_to_bytes
0.0   0.00   3.17   15190   0.0000  cache_read
0.0   0.00   3.17       2   0.      debug_file
0.0   0.00   3.17       2   0.      debug_mask
0.0   0.00   3.17       1   0.      debug_close
0.0   0.00   3.17       4   0.      audio_close
0.0   0.00   3.17       1   0.      dvi_adpcm_init_state
0.0   0.00   3.17       1   0.      dvi_adpcm_init
0.0   0.00   3.17       3   0.      g711_init
0.0   0.00   3.17       1   0.      G721_init
0.0   0.00   3.17       1   0.      G723_init
0.0   0.00   3.17       2   0.      gettimeofday_double
0.0   0.00   3.17      65   0.0     gettimeofday_ntp
0.0   0.00   3.17       1   0.      GSM_init
0.0   0.00   3.17       2   0.      List_to_index
0.0   0.00   3.17    6137   0.000   audio_vu
0.0   0.00   3.17   30652   0.0000  ring_left
0.0   0.00   3.17       2   0.      list_key2value
```

```
0.0   0.00   3.17       1   0.      lpc_init
0.0   0.00   3.17       6   0.      notify_set_input_func
0.0   0.00   3.17   30550   0.0000  m_free
0.0   0.00   3.17   15259   0.0000  m_freem
0.0   0.00   3.17       1   0.      m_create
0.0   0.00   3.17       1   0.      onlyone
0.0   0.00   3.17     127   0.00    rtcp_period
0.0   0.00   3.17   73984   0.0000  l14_to_pcmu
0.0   0.00   3.17   15354   0.0000  l16_to_pcmu
0.0   0.00   3.17       6   0.      peakmeter_trace
0.0   0.00   3.17       2   0.      peakmeter_init
0.0   0.00   3.17       4   0.      md_32
0.0   0.00   3.17       4   0.      random32
0.0   0.00   3.17      11   0.0     sd_trace
0.0   0.00   3.17       1   0.      sd_init
0.0   0.00   3.17   15190   0.0000  sd
0.0   0.00   3.17       1   0.      source_file
0.0   0.00   3.17       1   0.      strcpy_pad
0.0   0.00   3.17      14   0.0     strsaven
0.0   0.00   3.17       2   0.      tcl
0.0   0.00   3.17     133   0.00    timeval_ntp32
0.0   0.00   3.17      10   0.0     tsap_alloc
0.0   0.00   3.17       9   0.      tsap_free
0.0   0.00   3.17       2   0.      tsap_equ_a
0.0   0.00   3.17     432   0.00    tsap_ntoa
0.0   0.00   3.17       1   0.      tsap_gethostaddress
0.0   0.00   3.17       4   0.      tsap_getport
0.0   0.00   3.17       7   0.      tsap_setport
0.0   0.00   3.17       1   0.      tsap_gethostbyname
0.0   0.00   3.17       2   0.      tsap_sprintf
0.0   0.00   3.17       2   0.      tsap_ip4
0.0   0.00   3.17      11   0.0     htons
0.0   0.00   3.17       4   0.      ntohs
0.0   0.00   3.17       2   0.      UDP_connect
0.0   0.00   3.17       1   0.      Audio_init
0.0   0.00   3.17       2   0.      UDP_close
0.0   0.00   3.17       2   0.      itoa
0.0   0.00   3.17      11   0.0     ahw_open_control
0.0   0.00   3.17       2   0.      ahw_getdev
0.0   0.00   3.17       1   0.      l16_init
0.0   0.00   3.17       1   0.      ahw_close
0.0   0.00   3.17   15181   0.0000  ahw_set
0.0   0.00   3.17       1   0.      ahw_fmt
0.0   0.00   3.17     299   0.00    ahw_write_queue
0.0   0.00   3.17   15352   0.0000  ahw_write
0.0   0.00   3.17   28039   0.0000  ahw_read
0.0   0.00   3.17      30   0.0     exp10
0.0   0.00   3.17       4   0.      gethostid
0.0   0.00   3.17       2   0.      signal
0.0   0.00   3.17       4   0.      MD5Init
```

```
0.0    0.00    3.17        12    0.0       MD5Update
0.0    0.00    3.17         4    0.        MD5Final
0.0    0.00    3.17        84    0.0       MD5Transform
0.0    0.00    3.17         8    0.        Encode
0.0    0.00    3.17        84    0.0       Decode
0.0    0.00    3.17   15361    0.0000  tsap_cpy
0.0    0.00    3.17         1    0.        ahw_open
0.0    0.00    3.17         2    0.        cmd_terminate
```

APPENDIX 2:

Profile file of Nevot CPU consumption with ADPCM audio coding

```
%Time Seconds Cumsecs #Calls msec/call Name

 26.4   1.75   1.75   30654    0.0571  audio_stats
 25.9   1.72   3.47   15221    0.1130  dvi_adpcm_encode
 25.6   1.70   5.17   15276    0.1113  dvi_adpcm_decode
  3.9   0.26   5.43   15335    0.0170  mix2_pcmu
  1.7   0.11   5.54                    Tk_DoOneEvent
  1.2   0.08   5.62                    $2
  1.1   0.07   5.69                    _mcount
  1.1   0.07   5.76   15609    0.0045  member_find
  1.1   0.07   5.83   15543    0.0045  rx
  0.9   0.06   5.89   28383    0.0021  tx
  0.8   0.05   5.94   30700    0.0016  ring_mix
  0.8   0.05   5.99   15476    0.0032  play_local
  0.6   0.04   6.03                    TclParseWords
  0.6   0.04   6.07   43926    0.0009  file_handler
  0.6   0.04   6.11   15221    0.0026  tx_packet
  0.6   0.04   6.15                    StringFind
  0.5   0.03   6.18                    Tcl_ScanElement
  0.5   0.03   6.21   30697    0.0010  peakmeter
  0.5   0.03   6.24  131328    0.0002  pcmu_to_l14
  0.5   0.03   6.27   45983    0.0007  m_get
  0.5   0.03   6.30                    ValueToPixel
  0.5   0.03   6.33   30700    0.0010  bytes2samples
  0.3   0.02   6.35   73984    0.0003  l14_to_pcmu
  0.3   0.02   6.37   15284    0.0013  tx_packet_session
  0.3   0.02   6.39   15479    0.0013  rx_stats
  0.2   0.01   6.40                    Tcl_ConvertElement
  0.2   0.01   6.41                    DisplayVerticalMeter
  0.2   0.01   6.42    6451    0.002   event
  0.2   0.01   6.43                    Tcl_DStringInit
  0.2   0.01   6.44                    Tcl_Eval
  0.2   0.01   6.45   30700    0.0003  talking_set
  0.2   0.01   6.46   30649    0.0003  ring_cpy
  0.2   0.01   6.47   30748    0.0003  ring_action
  0.2   0.01   6.48   15287    0.0007  session_ismulticast
  0.2   0.01   6.49                    Tcl_DStringAppendElement
  0.2   0.01   6.50                    gcc2_compiled., strtod
  0.2   0.01   6.51   28383    0.0004  ahw_read
  0.2   0.01   6.52   15221    0.0007  agc
  0.2   0.01   6.53   15284    0.0007  tx_member
  0.2   0.01   6.54                    ExprGetValue
  0.2   0.01   6.55   15479    0.0006  rtp_read_data
  0.2   0.01   6.56   30497    0.0003  htons
  0.2   0.01   6.57   15221    0.0007  sd
```

```
0.2   0.01   6.58  15221     0.0007  rtp_write_data
0.2   0.01   6.59  15476     0.0006  talking_check
0.2   0.01   6.60  31367     0.0003  htonl
0.2   0.01   6.61                    Tdp_CommandTrace
0.2   0.01   6.62                    NewVar
0.2   0.01   6.63  15434     0.0006  l16_to_pcmu
0.0   0.00   6.63     63     0.0     rtp_write_control
0.0   0.00   6.63     64     0.0     rtp_read_sdes
0.0   0.00   6.63     64     0.0     rtp_read_traffic
0.0   0.00   6.63     64     0.0     rtp_read_control
0.0   0.00   6.63    558     0.00    rtcp_text
0.0   0.00   6.63     62     0.0     rtp_write_traffic
0.0   0.00   6.63  30700     0.0000  htons
0.0   0.00   6.63     62     0.0     rtp_write_sdes
0.0   0.00   6.63      3     0.      rtp_change_audio
0.0   0.00   6.63      2     0.      cmd_terminate
0.0   0.00   6.63      3     0.      rtp_init
0.0   0.00   6.63  15607     0.0000  ntohl
0.0   0.00   6.63  15433     0.0000  play_silence
0.0   0.00   6.63   6424     0.000   vu_show
0.0   0.00   6.63     62     0.0     rx_periodic
0.0   0.00   6.63  15479     0.0000  ntohs
0.0   0.00   6.63      2     0.      Audio_MemberClose
0.0   0.00   6.63     61     0.0     rtp_packets_exp
0.0   0.00   6.63      1     0.      Audio_SessionFind
0.0   0.00   6.63      2     0.      Audio_MemberListen
0.0   0.00   6.63      3     0.      talkspurt
0.0   0.00   6.63      1     0.      Audio_SessionSetTTL
0.0   0.00   6.63  15479     0.0000  rx_debug
0.0   0.00   6.63      2     0.      Audio_SessionSetAudio
0.0   0.00   6.63      2     0.      Audio_SessionTalk
0.0   0.00   6.63      1     0.      Audio_SessionListen
0.0   0.00   6.63      2     0.      Audio_MemberDelete
0.0   0.00   6.63      1     0.      Tcl_AppInit
0.0   0.00   6.63      2     0.      talking_delete
0.0   0.00   6.63      1     0.      rx_init
0.0   0.00   6.63      8     0.      rx_trace
0.0   0.00   6.63      1     0.      rx_config
0.0   0.00   6.63      4     0.      tx_trace
0.0   0.00   6.63      1     0.      tx_config
0.0   0.00   6.63     62     0.0     tx_periodic_audio
0.0   0.00   6.63  15476     0.0000  tx_local_audio
0.0   0.00   6.63      1     0.      session_open
0.0   0.00   6.63      2     0.      tx_start
0.0   0.00   6.63     59     0.0     tx_samples
0.0   0.00   6.63      1     0.      main
0.0   0.00   6.63      1     0.      session_add_host
0.0   0.00   6.63      1     0.      tx_packet_init
0.0   0.00   6.63      1     0.      tx_packet_close
0.0   0.00   6.63  15221     0.0000  tx_session_attach
```

```
0.0   0.00   6.63      1    0.      Audio_SessionDelete
0.0   0.00   6.63      2    0.      cmd_simple
0.0   0.00   6.63      1    0.      Audio_SessionAlloc
0.0   0.00   6.63      1    0.      Audio_SessionOpen
0.0   0.00   6.63      8    0.      Audio_SessionSetSDES
0.0   0.00   6.63      1    0.      Audio_SessionSetKey
0.0   0.00   6.63      3    0.      audio_change
0.0   0.00   6.63      1    0.      audio_out
0.0   0.00   6.63      1    0.      Audio_SessionSetDuplex
0.0   0.00   6.63      4    0.      audio_close
0.0   0.00   6.63    458    0.00    member_sdes
0.0   0.00   6.63   6424    0.000   audio_vu
0.0   0.00   6.63      8    0.      audio_chunk
0.0   0.00   6.63     17    0.0     cmd_member
0.0   0.00   6.63      1    0.      ring_create
0.0   0.00   6.63      2    0.      member_add
0.0   0.00   6.63     65    0.0     member_announce
0.0   0.00   6.63      3    0.      talking_clear
0.0   0.00   6.63      2    0.      ring_clear
0.0   0.00   6.63      8    0.      notify_set_input_func
0.0   0.00   6.63     61    0.0     timer_handler
0.0   0.00   6.63      1    0.      Audio_exit
0.0   0.00   6.63     62    0.0     tx_periodic
0.0   0.00   6.63      1    0.      cmd_session_audio
0.0   0.00   6.63      1    0.      tx_init
0.0   0.00   6.63      6    0.      agc_trace
0.0   0.00   6.63      1    0.      agc_init
0.0   0.00   6.63    140    0.00    audio_cmp
0.0   0.00   6.63    169    0.00    audio_descr
0.0   0.00   6.63      2    0.      audio_seconds_to_bytes
0.0   0.00   6.63  15436    0.0000  audio_silence
0.0   0.00   6.63      1    0.      audio_stats_init
0.0   0.00   6.63  15221    0.0000  cache_read
0.0   0.00   6.63      2    0.      debug_file
0.0   0.00   6.63      2    0.      debug_mask
0.0   0.00   6.63  59555    0.0000  debug
0.0   0.00   6.63      2    0.      dvi_adpcm_init_state
0.0   0.00   6.63      2    0.      dvi_adpcm_init
0.0   0.00   6.63      1    0.      vat_init
0.0   0.00   6.63      2    0.      g711_init
0.0   0.00   6.63      1    0.      cmd_init
0.0   0.00   6.63      1    0.      G721_init
0.0   0.00   6.63      1    0.      Audio_init
0.0   0.00   6.63  30756    0.0000  audio_open
0.0   0.00   6.63      3    0.      gettimeofday_double
0.0   0.00   6.63     59    0.0     gettimeofday_ntp
0.0   0.00   6.63      1    0.      GSM_init
0.0   0.00   6.63      1    0.      audio_close_idle
0.0   0.00   6.63      1    0.      l16_init
0.0   0.00   6.63      2    0.      list_to_index
```

```
0.0   0.00   6.63  31015   0.0000  ring_left
0.0   0.00   6.63      2   0.      list_key2value
0.0   0.00   6.63      1   0.      lpc_init
0.0   0.00   6.63     26   0.0     cmd_session
0.0   0.00   6.63     63   0.0     notify_set_periodic_func
0.0   0.00   6.63  15285   0.0000  m_freem
0.0   0.00   6.63      1   0.      m_create
0.0   0.00   6.63      1   0.      mix_init
0.0   0.00   6.63      1   0.      onlyone
0.0   0.00   6.63    126   0.00    rtcp_period
0.0   0.00   6.63      1   0.      pcmu_linear_init
0.0   0.00   6.63      6   0.      peakmeter_trace
0.0   0.00   6.63      2   0.      peakmeter_init
0.0   0.00   6.63      4   0.      cmd_general
0.0   0.00   6.63      4   0.      md_32
0.0   0.00   6.63      4   0.      random32
0.0   0.00   6.63     11   0.0     sd_trace
0.0   0.00   6.63      1   0.      sd_init
0.0   0.00   6.63      1   0.      source_file
0.0   0.00   6.63      1   0.      Strcpy_pad
0.0   0.00   6.63     14   0.0     strsaven
0.0   0.00   6.63      2   0.      tcl
0.0   0.00   6.63    123   0.00    timeval_ntp32
0.0   0.00   6.63     10   0.0     tsap_alloc
0.0   0.00   6.63      9   0.      tsap_free
0.0   0.00   6.63  15549   0.0000  tsap_cpy
0.0   0.00   6.63      2   0.      tsap_equ_a
0.0   0.00   6.63    460   0.00    tsap_ntoa
0.0   0.00   6.63  15292   0.0000  tsap_ismulticast
0.0   0.00   6.63      1   0.      tsap_gethostaddress
0.0   0.00   6.63      4   0.      tsap_getport
0.0   0.00   6.63      7   0.      tsap_setport
0.0   0.00   6.63      1   0.      tsap_gethostbyname
0.0   0.00   6.63      2   0.      tsap_sprintf
0.0   0.00   6.63      2   0.      tsap_ip4
0.0   0.00   6.63     11   0.0     htons
0.0   0.00   6.63      4   0.      ntohs
0.0   0.00   6.63      2   0.      UDP_connect
0.0   0.00   6.63  15543   0.0000  UDP_read
0.0   0.00   6.63  15284   0.0000  UDP_write
0.0   0.00   6.63      2   0.      UDP_close
0.0   0.00   6.63      2   0.      itoa
0.0   0.00   6.63     13   0.0     ahw_open_control
0.0   0.00   6.63      5   0.      ahw_getdev
0.0   0.00   6.63      2   0.      ahw_open
0.0   0.00   6.63      2   0.      ahw_close
0.0   0.00   6.63  15214   0.0000  ahw_set
0.0   0.00   6.63      3   0.      ahw_fmt
0.0   0.00   6.63    301   0.00    ahw_write_queue
0.0   0.00   6.63  15435   0.0000  ahw_write
```

```
0.0   0.00   6.63       1    0.      debug_close
0.0   0.00   6.63      30    0.0     exp10
0.0   0.00   6.63       4    0.      gethostid
0.0   0.00   6.63       2    0.      signal
0.0   0.00   6.63       4    0.      MD5Init
0.0   0.00   6.63      12    0.0     MD5Update
0.0   0.00   6.63       4    0.      MD5Final
0.0   0.00   6.63      84    0.0     MD5Transform
0.0   0.00   6.63       8    0.      Encode
0.0   0.00   6.63      84    0.0     Decode
0.0   0.00   6.63   15276    0.0000  ntohs
0.0   0.00   6.63       1    0.      G723_init
0.0   0.00   6.63   45981    0.0000  m_free
```

APPENDIX 3:

Profile file of Nevot CPU consumption with GSM audio coding

```
%Time Seconds Cumsecs #Calls msec/call Name

 19.6   2.90   2.90                     Fast_Calculation_of_the_LTP_parameters
 13.7   2.03   4.93                     Short_term_analysis_filtering
 12.5   1.85   6.78                     Short_term_synthesis_filtering
 11.3   1.67   8.45   30435     0.0549  audio_stats
  4.7   0.69   9.14                     Weighting_filter
  4.5   0.67   9.81                     Fast_Autocorrelation
  4.2   0.62  10.43 2440739     0.0003  l16_to_pcmu
  3.0   0.44  10.87                     Gsm_Preprocess
  2.0   0.30  11.17                     Long_term_analysis_filtering
  2.0   0.29  11.46   15219     0.0191  mix2_pcmu
  1.8   0.26  11.72   15159     0.0172  GSM_decode
  1.8   0.26  11.98   15137     0.0172  GSM_encode
  1.4   0.21  12.19                     Gsm_Long_Term_Synthesis_Filtering
  1.3   0.20  12.39                     _mcount
  1.2   0.18  12.57                     Postprocessing
  1.1   0.16  12.73                     APCM_inverse_quantization
  0.9   0.13  12.86   15352     0.0085  rx
  0.8   0.12  12.98                     Gsm_Coder
  0.7   0.10  13.08                     RPE_grid_positioning
  0.7   0.10  13.18   15137     0.0066  tx_packet
  0.6   0.09  13.27                     APCM_quantization
  0.6   0.09  13.36                     Tk_DoOneEvent
  0.5   0.08  13.44                     $2
  0.5   0.08  13.52                     Reflection_coefficients
  0.5   0.07  13.59                     gsm_encode
  0.3   0.05  13.64                     RPE_grid_selection
  0.3   0.05  13.69   27973     0.0018  tx
  0.3   0.05  13.74                     Quantization_and_coding
  0.3   0.05  13.79                     LARp_to_rp
  0.3   0.05  13.84                     gsm_decode
  0.3   0.04  13.88                     gsm_asl
  0.3   0.04  13.92                     Coefficients_27_39
  0.3   0.04  13.96  131328     0.0003  pcmu_to_l14
  0.2   0.03  13.99                     Gsm_Short_Term_Synthesis_Filter
  0.2   0.03  14.02                     Gsm_Decoder
  0.2   0.03  14.05                     Decoding_of_the_coded_Log_Area_Ratios
  0.2   0.03  14.08   30427     0.0010  peakmeter
  0.2   0.03  14.11   15292     0.0020  rx_stats
  0.2   0.03  14.14   15199     0.0020  tx_member
  0.2   0.03  14.17                     gsm_asr
  0.2   0.03  14.20   45628     0.0007  m_get
  0.2   0.03  14.23                     gsm_div
  0.1   0.02  14.25                     Tk_Release
```

```
0.1   0.02  14.27  15137   0.0013  cache_read
0.1   0.02  14.29      1  20.      audio_stats_init
0.1   0.02  14.31                  Tcl_Eval
0.1   0.02  14.33  43325   0.0005  file_handler
0.1   0.02  14.35  30429   0.0007  ring_mix
0.1   0.02  14.37  73984   0.0003  l14_to_pcmu
0.1   0.02  14.39  30642   0.0007  ring_left
0.1   0.02  14.41  30397   0.0007  ring_cpy
0.1   0.02  14.43  30429   0.0007  bytes2samples
0.1   0.02  14.45  15199   0.0013  tx_packet_session
0.1   0.02  14.47                  Coefficients_13_26
0.1   0.02  14.49  15290   0.0013  tx_local_audio
0.1   0.02  14.51                  Gsm_RPE_Decoding
0.1   0.01  14.52  30429   0.0003  htons
0.1   0.01  14.53   6318   0.002   event
0.1   0.01  14.54                  $1
0.1   0.01  14.55  15292   0.0007  rtp_read_data
0.1   0.01  14.56  45626   0.0002  m_free
0.1   0.01  14.57  15299   0.0007  ahw_write
0.1   0.01  14.58                  SetMeterValue
0.1   0.01  14.59                  Tcl_ScanElement
0.1   0.01  14.60                  StringFind
0.1   0.01  14.61  15290   0.0007  play_local
0.1   0.01  14.62                  Tcl_GetDouble
0.1   0.01  14.63                  ExprLex
0.1   0.01  14.64                  gsm_norm
0.1   0.01  14.65  15202   0.0007  session_ismulticast
0.1   0.01  14.66                  Gsm_LPC_Analysis
0.1   0.01  14.67                  Transformation_to_Log_Area_Ratios
0.1   0.01  14.68  15130   0.0007  ahw_set
0.1   0.01  14.69                  ValueToPixel
0.1   0.01  14.70  15137   0.0007  sd
0.1   0.01  14.71    121   0.08    rtcp_period
0.1   0.01  14.72                  Tcl_DStringAppendElement
0.1   0.01  14.73                  TclParseWords
0.1   0.01  14.74                  Gsm_Short_Term_Analysis_Filter
0.1   0.01  14.75                  NewVar
0.1   0.01  14.76                  gcc2_compiled., strtod
0.1   0.01  14.77                  Tdp_CommandTrace
0.1   0.01  14.78  30429   0.0003  talking_set
0.1   0.01  14.79                  RoundToResolution
0.1   0.01  14.80  15207   0.0007  tsap_ismulticast
0.1   0.01  14.81  15352   0.0007  UDP_read
0.1   0.01  14.82  27973   0.0004  ahw_read
0.1   0.01  14.83                  gsm_sub
0.0   0.00  14.83      2   0.      Audio_SessionTalk
0.0   0.00  14.83      1   0.      Audio_SessionListen
0.0   0.00  14.83  15290   0.0000  talking_check
0.0   0.00  14.83      2   0.      talking_delete
0.0   0.00  14.83      2   0.      talking_clear
```

```
0.0   0.00  14.83      2   0.      Audio_SessionSetAudio
0.0   0.00  14.83      8   0.      rx_trace
0.0   0.00  14.83      1   0.      Audio_SessionSetDuplex
0.0   0.00  14.83      1   0.      Audio_exit
0.0   0.00  14.83      1   0.      Audio_SessionSetKey
0.0   0.00  14.83      1   0.      rx_config
0.0   0.00  14.83  15292   0.0000  rx_debug
0.0   0.00  14.83     61   0.0     tx_periodic
0.0   0.00  14.83      1   0.      tx_start
0.0   0.00  14.83      4   0.      tx_trace
0.0   0.00  14.83      2   0.      talkspurt
0.0   0.00  14.83      1   0.      tx_config
0.0   0.00  14.83      1   0.      tx_packet_init
0.0   0.00  14.83      1   0.      Audio_SessionSetTTL
0.0   0.00  14.83      8   0.      Audio_SessionSetSDES
0.0   0.00  14.83      1   0.      Audio_SessionOpen
0.0   0.00  14.83      1   0.      Audio_SessionAlloc
0.0   0.00  14.83      1   0.      Audio_SessionDelete
0.0   0.00  14.83      1   0.      vat_init
0.0   0.00  14.83     61   0.0     tx_periodic_audio
0.0   0.00  14.83      1   0.      audio_out
0.0   0.00  14.83     59   0.0     tx_samples
0.0   0.00  14.83      4   0.      audio_close
0.0   0.00  14.83      1   0.      audio_close_idle
0.0   0.00  14.83   6291   0.000   audio_vu
0.0   0.00  14.83      8   0.      audio_chunk
0.0   0.00  14.83  15412   0.0000  ntohl
0.0   0.00  14.83      1   0.      ring_create
0.0   0.00  14.83      1   0.      Audio_SessionFind
0.0   0.00  14.83      1   0.      session_open
0.0   0.00  14.83      3   0.      rtp_init
0.0   0.00  14.83      2   0.      ring_clear
0.0   0.00  14.83      3   0.      rtp_change_audio
0.0   0.00  14.83      1   0.      session_add_host
0.0   0.00  14.83      1   0.      tx_init
0.0   0.00  14.83      1   0.      tx_packet_close
0.0   0.00  14.83     62   0.0     notify_set_periodic_func
0.0   0.00  14.83      1   0.      rx_init
0.0   0.00  14.83     61   0.0     rx_periodic
0.0   0.00  14.83  15137   0.0000  tx_session_attach
0.0   0.00  14.83  15298   0.0000  play_silence
0.0   0.00  14.83  15292   0.0000  ntohs
0.0   0.00  14.83  31084   0.0000  htonl
0.0   0.00  14.83      6   0.      agc_trace
0.0   0.00  14.83      3   0.      audio_change
0.0   0.00  14.83  15137   0.0000  agc
0.0   0.00  14.83  30452   0.0000  audio_open
0.0   0.00  14.83    169   0.00    audio_descr
0.0   0.00  14.83  30477   0.0000  ring_action
0.0   0.00  14.83  15301   0.0000  audio_silence
```

```
0.0   0.00  14.83     60   0.0     rtp_read_traffic
0.0   0.00  14.83     60   0.0     rtp_read_sdes
0.0   0.00  14.83  15137   0.0000  rtp_write_data
0.0   0.00  14.83      2   0.      debug_file
0.0   0.00  14.83      2   0.      debug_mask
0.0   0.00  14.83      1   0.      debug_close
0.0   0.00  14.83  58839   0.0000  debug
0.0   0.00  14.83      1   0.      dvi_adpcm_init_state
0.0   0.00  14.83      1   0.      dvi_adpcm_init
0.0   0.00  14.83     60   0.0     rtp_read_control
0.0   0.00  14.83      1   0.      G721_init
0.0   0.00  14.83      1   0.      G723_init
0.0   0.00  14.83      6   0.      notify_set_input_func
0.0   0.00  14.83     59   0.0     gettimeofday_ntp
0.0   0.00  14.83     60   0.0     timer_handler
0.0   0.00  14.83     62   0.0     rtp_write_control
0.0   0.00  14.83     61   0.0     rtp_write_traffic
0.0   0.00  14.83      1   0.      l16_init
0.0   0.00  14.83      2   0.      list_to_index
0.0   0.00  14.83      1   0.      lpc_init
0.0   0.00  14.83     61   0.0     rtp_write_sdes
0.0   0.00  14.83  15200   0.0000  m_freem
0.0   0.00  14.83      1   0.      m_create
0.0   0.00  14.83     60   0.0     rtp_packets_exp
0.0   0.00  14.83      1   0.      mix_init
0.0   0.00  14.83      2   0.      Audio_MemberListen
0.0   0.00  14.83      1   0.      pcmu_linear_init
0.0   0.00  14.83  15414   0.0000  member_find
0.0   0.00  14.83      6   0.      peakmeter_trace
0.0   0.00  14.83      2   0.      peakmeter_init
0.0   0.00  14.83      4   0.      md_32
0.0   0.00  14.83      4   0.      random32
0.0   0.00  14.83     11   0.0     sd_trace
0.0   0.00  14.83      1   0.      sd_init
0.0   0.00  14.83      1   0.      source_file
0.0   0.00  14.83     14   0.0     strsaven
0.0   0.00  14.83      2   0.      tcl
0.0   0.00  14.83    121   0.00    timeval_ntp32
0.0   0.00  14.83     10   0.0     tsap_alloc
0.0   0.00  14.83      9   0.      tsap_free
0.0   0.00  14.83  15358   0.0000  tsap_cpy
0.0   0.00  14.83      2   0.      tsap_equ_a
0.0   0.00  14.83    432   0.00    tsap_ntoa
0.0   0.00  14.83      1   0.      agc_init
0.0   0.00  14.83      1   0.      tsap_gethostaddress
0.0   0.00  14.83      4   0.      tsap_getport
0.0   0.00  14.83      7   0.      tsap_setport
0.0   0.00  14.83      1   0.      tsap_gethostbyname
0.0   0.00  14.83      2   0.      tsap_sprintf
0.0   0.00  14.83      2   0.      tsap_ip4
```

```
0.0   0.00  14.83      11   0.0     htons
0.0   0.00  14.83       4   0.      ntohs
0.0   0.00  14.83       2   0.      UDP_connect
0.0   0.00  14.83     136   0.00    audio_cmp
0.0   0.00  14.83   15199   0.0000  UDP_write
0.0   0.00  14.83       2   0.      UDP_close
0.0   0.00  14.83       2   0.      itoa
0.0   0.00  14.83      13   0.0     ahw_open_control
0.0   0.00  14.83       4   0.      ahw_getdev
0.0   0.00  14.83       1   0.      ahw_open
0.0   0.00  14.83       1   0.      ahw_close
0.0   0.00  14.83       2   0.      audio_seconds_to_bytes
0.0   0.00  14.83       3   0.      ahw_fmt
0.0   0.00  14.83     298   0.00    ahw_write_queue
0.0   0.00  14.83      30   0.0     exp10
0.0   0.00  14.83       4   0.      gethostid
0.0   0.00  14.83       2   0.      signal
0.0   0.00  14.83       4   0.      MD5Init
0.0   0.00  14.83      12   0.0     MD5Update
0.0   0.00  14.83       4   0.      MD5Final
0.0   0.00  14.83      84   0.0     MD5Transform
0.0   0.00  14.83       8   0.      Encode
0.0   0.00  14.83      84   0.0     Decode
0.0   0.00  14.83       2   0.      Audio_MemberDelete
0.0   0.00  14.83       2   0.      Audio_MemberClose
0.0   0.00  14.83       2   0.      member_add
0.0   0.00  14.83      61   0.0     member_announce
0.0   0.00  14.83     430   0.00    member_sdes
0.0   0.00  14.83       1   0.      Audio_init
0.0   0.00  14.83       1   0.      cmd_init
0.0   0.00  14.83       2   0.      cmd_terminate
0.0   0.00  14.83       2   0.      cmd_simple
0.0   0.00  14.83       2   0.      g711_init
0.0   0.00  14.83      17   0.0     cmd_member
0.0   0.00  14.83       2   0.      gettimeofday_double
0.0   0.00  14.83       1   0.      cmd_session_audio
0.0   0.00  14.83      26   0.0     cmd_session
0.0   0.00  14.83       4   0.      cmd_general
0.0   0.00  14.83       1   0.      Tcl_AppInit
0.0   0.00  14.83    6291   0.000   vu_show
0.0   0.00  14.83       1   0.      main
0.0   0.00  14.83       2   0.      GSM_init
0.0   0.00  14.83       2   0.      list_key2value
0.0   0.00  14.83     549   0.00    rtcp_text
0.0   0.00  14.83       1   0.      onlyone
0.0   0.00  14.83       1   0.      strcpy_pad
```

APPENDIX 4:

Profile file of Nevot CPU consumption with LPC audio coding

```
%Time Seconds Cumsecs #Calls msec/call Name
 36.3   9.16    9.16   14967    0.6120  lpc_decode_frame
 35.6   8.98   18.14   45360    0.1980  auto_correl
  6.9   1.74   19.88   15120    0.1151  lpc_encode_frame
  6.7   1.70   21.58   30175    0.0563  audio_stats
  3.6   0.90   22.48   15120    0.0595  inverse_filter
  2.8   0.70   23.18 2409776    0.0003  l16_to_pcmu
  1.8   0.46   23.64   30240    0.0152  durbin
  0.8   0.21   23.85   15041    0.0140  mix2_pcmu
  0.6   0.15   24.00                    _mcount
  0.6   0.14   24.14                    Tk_DoOneEvent
  0.4   0.11   24.25   15120    0.0073  calc_pitch
  0.3   0.07   24.32   14968    0.0047  rx
  0.3   0.07   24.39   30087    0.0023  ring_mix
  0.3   0.07   24.46   15120    0.0046  agc
  0.2   0.06   24.52   15120    0.0040  tx_packet
  0.2   0.05   24.57   30087    0.0017  bytes2samples
  0.2   0.04   24.61   15120    0.0026  tx_local_audio
  0.1   0.03   24.64   14967    0.0020  lpc_decode
  0.1   0.03   24.67  131328    0.0002  pcmu_to_l14
  0.1   0.03   24.70   73984    0.0004  l14_to_pcmu
  0.1   0.03   24.73                    TclParseWords
  0.1   0.03   24.76   15122    0.0020  tx_packet_session
  0.1   0.03   24.79   27634    0.0011  tx
  0.1   0.02   24.81                    DisplayVerticalMeter
  0.1   0.02   24.83                    Tcl_ConvertElement
  0.1   0.02   24.85   29995    0.0007  ring_action
  0.1   0.02   24.87                    $2
  0.1   0.02   24.89   14974    0.0013  tsap_cpy
  0.1   0.02   24.91   14968    0.0013  UDP_read
  0.1   0.02   24.93   15120    0.0013  play_local
  0.1   0.02   24.95                    Tcl_Eval
  0.1   0.02   24.97   14967    0.0013  rx_stats
  0.1   0.02   24.99   15120    0.0013  rtp_write_data
  0.1   0.02   25.01   14971    0.0013  member_find
  0.0   0.01   25.02                    Tcl_GetInt, gcc2_compiled.
  0.0   0.01   25.03                    Tdp_CreateAddress
  0.0   0.01   25.04                    InterpProc
  0.0   0.01   25.05      84    0.1     MD5Transform
  0.0   0.01   25.06   15120    0.0007  cache_read
  0.0   0.01   25.07   29963    0.0003  ring_cpy
  0.0   0.01   25.08                    Tk_HandleEvent
  0.0   0.01   25.09   15120    0.0007  talking_check
  0.0   0.01   25.10                    Tcl_ScanElement
```

```
0.0   0.01  25.11  29935    0.0003  audio_open
0.0   0.01  25.12  15113    0.0007  ahw_set
0.0   0.01  25.13  15122    0.0007  tx_member
0.0   0.01  25.14  15122    0.0007  UDP_write
0.0   0.01  25.15  14967    0.0007  rx_debug
0.0   0.01  25.16  14967    0.0007  rtp_read_data
0.0   0.01  25.17  30240    0.0003  peakmeter
0.0   0.01  25.18                   Tcl_SetVar2
0.0   0.01  25.19  15125    0.0007  session_ismulticast
0.0   0.01  25.20  15120    0.0007  lpc_encode
0.0   0.01  25.21                   gcc2_compiled., strtoul
0.0   0.01  25.22  27634    0.0004  ahw_read
0.0   0.01  25.23  15058    0.0007  audio_silence
0.0   0.01  25.24                   Tdp_CommandTrace
0.0   0.00  25.24  30087    0.0000  htons
0.0   0.00  25.24  14967    0.0000  ntohl
0.0   0.00  25.24  14967    0.0000  ntohs
0.0   0.00  25.24      3    0.      talkspurt
0.0   0.00  25.24      2    0.      member_add
0.0   0.00  25.24      1    0.      rtp_read_control
0.0   0.00  25.24  30087    0.0000  htonl
0.0   0.00  25.24      1    0.      rx_periodic
0.0   0.00  25.24      1    0.      rx_init
0.0   0.00  25.24      3    0.      rtp_change_audio
0.0   0.00  25.24      3    0.      rtp_init
0.0   0.00  25.24      2    0.      member_announce
0.0   0.00  25.24      1    0.      rtp_read_traffic
0.0   0.00  25.24      1    0.      rtp_read_sdes
0.0   0.00  25.24      2    0.      Audio_MemberClose
0.0   0.00  25.24      1    0.      Audio_SessionSetTTL
0.0   0.00  25.24      1    0.      Audio_SessionSetKey
0.0   0.00  25.24      1    0.      Audio_SessionSetDuplex
0.0   0.00  25.24      2    0.      Audio_SessionTalk
0.0   0.00  25.24      2    0.      rtp_write_control
0.0   0.00  25.24      1    0.      rtp_write_traffic
0.0   0.00  25.24      1    0.      rtp_write_sdes
0.0   0.00  25.24      9    0.      rtcp_text
0.0   0.00  25.24  15055    0.0000  play_silence
0.0   0.00  25.24     17    0.0     member_sdes
0.0   0.00  25.24      8    0.      rx_trace
0.0   0.00  25.24      1    0.      rx_config
0.0   0.00  25.24      4    0.      tx_trace
0.0   0.00  25.24      1    0.      session_open
0.0   0.00  25.24      1    0.      tx_periodic_audio
0.0   0.00  25.24      1    0.      Audio_init
0.0   0.00  25.24      1    0.      session_add_host
0.0   0.00  25.24      1    0.      tx_start
0.0   0.00  25.24      1    0.      Audio_SessionFind
0.0   0.00  25.24      1    0.      tx_init
0.0   0.00  25.24      1    0.      Audio_SessionDelete
```

```
0.0   0.00  25.24      1   0.      tx_packet_close
0.0   0.00  25.24  15120   0.0000  tx_session_attach
0.0   0.00  25.24      1   0.      Audio_SessionAlloc
0.0   0.00  25.24   3457   0.000   event
0.0   0.00  25.24      1   0.      Audio_SessionOpen
0.0   0.00  25.24      8   0.      Audio_SessionSetSDES
0.0   0.00  25.24      2   0.      Audio_SessionSetAudio
0.0   0.00  25.24      1   0.      vat_init
0.0   0.00  25.24      2   0.      Audio_MemberListen
0.0   0.00  25.24      1   0.      Audio_exit
0.0   0.00  25.24      1   0.      Audio_SessionListen
0.0   0.00  25.24      4   0.      audio_close
0.0   0.00  25.24      1   0.      audio_close_idle
0.0   0.00  25.24  30087   0.0000  talking_set
0.0   0.00  25.24   3426   0.000   audio_vu
0.0   0.00  25.24      2   0.      talking_delete
0.0   0.00  25.24      2   0.      cmd_simple
0.0   0.00  25.24      1   0.      ring_create
0.0   0.00  25.24     21   0.0     cmd_member
0.0   0.00  25.24      2   0.      talking_clear
0.0   0.00  25.24  30242   0.0000  ring_left
0.0   0.00  25.24      1   0.      cmd_session_audio
0.0   0.00  25.24      2   0.      Audio_MemberDelete
0.0   0.00  25.24      1   0.      tx_config
0.0   0.00  25.24      2   0.      notify_set_periodic_func
0.0   0.00  25.24      1   0.      tx_periodic
0.0   0.00  25.24      1   0.      cmd_init
0.0   0.00  25.24      1   0.      tx_packet_init
0.0   0.00  25.24      2   0.      cmd_terminate
0.0   0.00  25.24      6   0.      agc_trace
0.0   0.00  25.24      1   0.      agc_init
0.0   0.00  25.24     26   0.0     cmd_session
0.0   0.00  25.24    169   0.00    audio_descr
0.0   0.00  25.24      2   0.      audio_seconds_to_bytes
0.0   0.00  25.24      1   0.      audio_stats_init
0.0   0.00  25.24      2   0.      debug_file
0.0   0.00  25.24      2   0.      debug_mask
0.0   0.00  25.24      1   0.      debug_close
0.0   0.00  25.24  57748   0.0000  debug
0.0   0.00  25.24      1   0.      dvi_adpcm_init_state
0.0   0.00  25.24      1   0.      dvi_adpcm_init
0.0   0.00  25.24      2   0.      g711_init
0.0   0.00  25.24      3   0.      audio_change
0.0   0.00  25.24      1   0.      audio_out
0.0   0.00  25.24      4   0.      cmd_general
0.0   0.00  25.24      2   0.      gettimeofday_double
0.0   0.00  25.24      1   0.      GSM_init
0.0   0.00  25.24      8   0.      audio_chunk
0.0   0.00  25.24      2   0.      list_to_index
0.0   0.00  25.24      2   0.      ring_clear
```

```
0.0   0.00  25.24      2   0.      list_key2value
0.0   0.00  25.24      1   0.      Tcl_AppInit
0.0   0.00  25.24   3426   0.000   vu_show
0.0   0.00  25.24      2   0.      lpc_init
0.0   0.00  25.24      1   0.      main
0.0   0.00  25.24  42602   0.0000  file_handler
0.0   0.00  25.24      6   0.      notify_set_input_func
0.0   0.00  25.24  45364   0.0000  m_get
0.0   0.00  25.24  15123   0.0000  m_freem
0.0   0.00  25.24      1   0.      m_create
0.0   0.00  25.24      1   0.      mix_init
0.0   0.00  25.24      1   0.      onlyone
0.0   0.00  25.24      2   0.      rtcp_period
0.0   0.00  25.24      1   0.      pcmu_linear_init
0.0   0.00  25.24      6   0.      peakmeter_trace
0.0   0.00  25.24      2   0.      peakmeter_init
0.0   0.00  25.24      4   0.      md_32
0.0   0.00  25.24      4   0.      random32
0.0   0.00  25.24     11   0.0     sd_trace
0.0   0.00  25.24      1   0.      sd_init
0.0   0.00  25.24  15120   0.0000  sd
0.0   0.00  25.24      1   0.      source_file
0.0   0.00  25.24      1   0.      strcpy_pad
0.0   0.00  25.24     14   0.0     strsaven
0.0   0.00  25.24      2   0.      tcl
0.0   0.00  25.24      1   0.      timeval_ntp32
0.0   0.00  25.24     10   0.0     tsap_alloc
0.0   0.00  25.24      9   0.      tsap_free
0.0   0.00  25.24      2   0.      tsap_equ_a
0.0   0.00  25.24     19   0.0     tsap_ntoa
0.0   0.00  25.24  15130   0.0000  tsap_ismulticast
0.0   0.00  25.24      1   0.      tsap_gethostaddress
0.0   0.00  25.24      4   0.      tsap_getport
0.0   0.00  25.24      7   0.      tsap_setport
0.0   0.00  25.24      1   0.      tsap_gethostbyname
0.0   0.00  25.24      2   0.      tsap_sprintf
0.0   0.00  25.24      2   0.      tsap_ip4
0.0   0.00  25.24     11   0.0     htons
0.0   0.00  25.24      4   0.      ntohs
0.0   0.00  25.24      2   0.      UDP_connect
0.0   0.00  25.24      2   0.      UDP_close
0.0   0.00  25.24      2   0.      itoa
0.0   0.00  25.24     13   0.0     ahw_open_control
0.0   0.00  25.24      4   0.      ahw_getdev
0.0   0.00  25.24      1   0.      ahw_open
0.0   0.00  25.24      1   0.      ahw_close
0.0   0.00  25.24      3   0.      ahw_fmt
0.0   0.00  25.24    294   0.00    ahw_write_queue
0.0   0.00  25.24  15058   0.0000  ahw_write
0.0   0.00  25.24    144   0.00    audio_cmp
```

```
0.0   0.00   25.24      30     0.0      exp10
0.0   0.00   25.24       4     0.       gethostid
0.0   0.00   25.24       2     0.       signal
0.0   0.00   25.24       4     0.       MD5Init
0.0   0.00   25.24      12     0.0      MD5Update
0.0   0.00   25.24       4     0.       MD5Final
0.0   0.00   25.24       8     0.       Encode
0.0   0.00   25.24      84     0.0      Decode
0.0   0.00   25.24       1     0.       l16_init
0.0   0.00   25.24       1     0.       G721_init
0.0   0.00   25.24       1     0.       G723_init
0.0   0.00   25.24   45362     0.0000   m_free
```

APPENDIX 5:

Recorded debugging file of the received RTP-headers

```
901004499.738370 audio opened
901004499.739854 delay.multiplier=4 ring.next=12589920 ring.size=32768
901004499.739938 T+l  timestamp      seqno        t      delay         p      slack    d_avg    d_var      d_est
901004499.740012 T+l 2734872318     65011   12589920 1572684898   12589920         0      800       200 4294967295
901004499.749683  +l 2734872478     65012   12589920 1572684898   12589920         0      760       228 1572684898
901004499.769639  +l 2734872638     65013   12590080 1572684898   12590080         0      722       253 1572684738
901004499.789523  +l 2734872798     65014   12590240 1572684898   12590240         0      686       274 1572684738
901004499.809606  +l 2734872958     65015   12590400 1572684898   12590400         0      652       293 1572684738
901004499.820256 audio underflow (0)
901004499.829539  +l 2734873118     65016   12590560 1572684898   12590560         0      620       309 1572684738
901004499.849542  +l 2734873278     65017   12590720 1572684898   12590720         0      589       323 1572684738
901004499.869532  +l 2734873438     65018   12590880 1572684898   12590880         0      560       335 1572684738
901004499.889511  +l 2734873598     65019   12591040 1572684898   12591040         0      532       345 1572684738
901004499.909500  +l 2734873758     65020   12591200 1572684898   12591200         0      505       353 1572684738
901004499.929509  +l 2734873918     65021   12591360 1572684898   12591360         0      480       359 1572684738
901004499.949499  +l 2734874078     65022   12591520 1572684898   12591520         0      456       364 1572684738
901004499.969494  +l 2734874238     65023   12591680 1572684898   12591680         0      433       367 1572684738
901004499.989491  +l 2734874398     65024   12591840 1572684898   12591840         0      412       370 1572684738
901004500.009495  +l 2734874558     65025   12592000 1572684898   12592000         0      391       371 1572684738
901004500.029482  +l 2734874718     65026   12592160 1572684898   12592160         0      372       371 1572684738
901004500.049486  +l 2734874878     65027   12592320 1572684898   12592320         0      353       370 1572684738
901004500.069477  +l 2734875038     65028   12592480 1572684898   12592480         0      336       368 1572684738
901004500.089472  +l 2734875198     65029   12592640 1572684898   12592640         0      319       366 1572684738
901004500.109478  +l 2734875358     65030   12592800 1572684898   12592800         0      303       363 1572684738
901004500.129467  +l 2734875518     65031   12592960 1572684898   12592960         0      288       359 1572684738
901004500.149463  +l 2734875678     65032   12593120 1572684898   12593120         0      274       355 1572684738
901004500.169467  +l 2734875838     65033   12593280 1572684898   12593280         0      260       350 1572684738
901004500.189453  +l 2734875998     65034   12593440 1572684898   12593440         0      247       345 1572684738
901004500.209516  +l 2734876158     65035   12593600 1572684898   12593600         0      235       339 1572684738
901004500.229459  +l 2734876318     65036   12593760 1572684898   12593760         0      223       334 1572684738
901004500.249448  +l 2734876478     65037   12593920 1572684898   12593920         0      212       328 1572684738
901004500.269439  +l 2734876638     65038   12594080 1572684898   12594080         0      201       321 1572684738
901004500.289437  +l 2734876798     65039   12594240 1572684898   12594240         0      191       315 1572684738
901004500.309435  +l 2734876958     65040   12594400 1572684898   12594400         0      182       308 1572684738
901004500.329422  +l 2734877118     65041   12594560 1572684898   12594560         0      173       301 1572684738
901004500.349439  +l 2734877278     65042   12594720 1572684898   12594720         0      164       295 1572684738
901004500.369425  +l 2734877438     65043   12594880 1572684898   12594880         0      156       288 1572684738
901004500.389411  +l 2734877598     65044   12595040 1572684898   12595040         0      148       281 1572684738
901004500.409483  +l 2734877758     65045   12595200 1572684898   12595200         0      141       274 1572684738
901004500.429551  +l 2734877918     65046   12595360 1572684898   12595360         0      134       267 1572684738
901004500.449422  +l 2734878078     65047   12595520 1572684898   12595520         0      127       260 1572684738
901004500.469415  +l 2734878238     65048   12595680 1572684898   12595680         0      121       253 1572684738
901004500.489413  +l 2734878398     65049   12595840 1572684898   12595840         0      115       246 1572684738
901004500.509389  +l 2734878558     65050   12596000 1572684898   12596000         0      109       239 1572684738
901004500.529393  +l 2734878718     65051   12596160 1572684898   12596160         0      104       233 1572684738
901004500.549387  +l 2734878878     65052   12596320 1572684898   12596320         0       99      226 1572684738
901004500.569382  +l 2734879038     65053   12596480 1572684898   12596480         0       94      219 1572684738
901004500.589382  +l 2734879198     65054   12596640 1572684898   12596640         0       89      213 1572684738
901004500.609388  +l 2734879358     65055   12596800 1572684898   12596800         0       85      206 1572684738
901004500.629365  +l 2734879518     65056   12596960 1572684898   12596960         0       80      200 1572684738
901004500.649373  +l 2734879678     65057   12597120 1572684898   12597120         0       76      194 1572684738
901004500.669370  +l 2734879838     65058   12597280 1572684898   12597280         0       73      188 1572684738
901004500.689363  +l 2734879998     65059   12597440 1572684898   12597440         0       69      182 1572684738
901004500.709361  +l 2734880158     65060   12597600 1572684898   12597600         0       66      176 1572684738
```

```
901004500.729357  +l 2734880318      65061   12597760 1572684898   12597760         0       62       171 1572684738
901004500.749351  +l 2734880478      65062   12597920 1572684898   12597920         0       59       165 1572684738
901004500.769374  +l 2734880638      65063   12598080 1572684898   12598080         0       56       160 1572684738
901004500.789349  +l 2734880798      65064   12598240 1572684898   12598240         0       53       154 1572684738
901004500.809416  +l 2734880958      65065   12598400 1572684898   12598400         0       51       149 1572684738
901004500.819931 audio underflow (160)
901004500.829345  +l 2734881118      65066   12598560 1572684898   12598560         0       48       144 1572684738
901004500.849344  +l 2734881278      65067   12598720 1572684898   12598720         0       46       139 1572684738
……

901004510.930978  +l 2734961918         35   12679360 1572684898   12679360       0      0       0 1572684738
901004510.947621  +l 2734962078         36   12679520 1572684898   12679520       0      0       0 1572684738
901004510.967431  +l 2734962238         37   12679680 1572684898   12679680       0      0       0 1572684738
901004510.987828  +l 2734962398         38   12679840 1572684898   12679840       0      0       0 1572684738
901004510.998093 audio underflow (320)
901004511.007437  +l 2734962558         39   12680000 1572684898   12680000       0      0       0 1572684738
901004511.027412  +l 2734962718         40   12680160 1572684898   12680160       0      0       0 1572684738
901004511.047384  +l 2734962878         41   12680320 1572684898   12680320       0      0       0 1572684738
901004511.067360  +l 2734963038         42   12680480 1572684898   12680480       0      0       0 1572684738
901004511.087416  +l 2734963198         43   12680640 1572684898   12680640       0      0       0 1572684738
901004511.107409  +l 2734963358         44   12680800 1572684898   12680800       0      0       0 1572684738
901004511.127379  +l 2734963518         45   12680960 1572684898   12680960       0      0       0 1572684738
901004511.147349  +l 2734963678         46   12681120 1572684898   12681120       0      0       0 1572684738
901004511.167383  +l 2734963838         47   12681280 1572684898   12681280       0      0       0 1572684738
901004511.187332  +l 2734963998         48   12681440 1572684898   12681440       0      0       0 1572684738
901004511.207362  +l 2734964158         49   12681600 1572684898   12681600       0      0       0 1572684738
901004511.227330  +l 2734964318         50   12681760 1572684898   12681760       0      0       0 1572684738
901004511.247348  +l 2734964478         51   12681920 1572684898   12681920       0      0       0 1572684738
901004511.267375  +l 2734964638         52   12682080 1572684898   12682080       0      0       0 1572684738
901004511.287349  +l 2734964798         53   12682240 1572684898   12682240       0      0       0 1572684738
901004511.307310  +l 2734964958         54   12682400 1572684898   12682400       0      0       0 1572684738
901004511.328600  +l 2734965118         55   12682560 1572684898   12682560       0      0       0 1572684738
901004511.347315  +l 2734965278         56   12682720 1572684898   12682720       0      0       0 1572684738
901004511.367681  +l 2734965438         57   12682880 1572684898   12682880       0      0       0 1572684738
901004511.387309  +l 2734965598         58   12683040 1572684898   12683040       0      0       0 1572684738
901004511.407291  +l 2734965758         59   12683200 1572684898   12683200       0      0       0 1572684738
901004511.427289  +l 2734965918         60   12683360 1572684898   12683360       0      0       0 1572684738
901004511.447293  +l 2734966078         61   12683520 1572684898   12683520       0      0       0 1572684738
901004511.467286  +l 2734966238         62   12683680 1572684898   12683680       0      0       0 1572684738
901004511.487378  +l 2734966398         63   12683840 1572684898   12683840       0      0       0 1572684738
901004511.507333  +l 2734966558         64   12684000 1572684898   12684000       0      0       0 1572684738
901004511.527282  +l 2734966718         65   12684160 1572684898   12684160       0      0       0 1572684738
901004511.547266  +l 2734966878         66   12684320 1572684898   12684320       0      0       0 1572684738
901004511.567265  +l 2734967038         67   12684480 1572684898   12684480       0      0       0 1572684738
901004511.587321  +l 2734967198         68   12684640 1572684898   12684640       0      0       0 1572684738
901004511.607245  +l 2734967358         69   12684800 1572684898   12684800       0      0       0 1572684738
901004511.627283  +l 2734967518         70   12684960 1572684898   12684960       0      0       0 1572684738
901004511.647251  +l 2734967678         71   12685120 1572684898   12685120       0      0       0 1572684738
901004511.667266  +l 2734967838         72   12685280 1572684898   12685280       0      0       0 1572684738
901004511.687249  +l 2734967998         73   12685440 1572684898   12685440       0      0       0 1572684738
901004511.707237  +l 2734968158         74   12685600 1572684898   12685600       0      0       0 1572684738
901004511.727242  +l 2734968318         75   12685760 1572684898   12685760       0      0       0 1572684738
901004511.747224  +l 2734968478         76   12685920 1572684898   12685920       0      0       0 1572684738
901004511.767238  +l 2734968638         77   12686080 1572684898   12686080       0      0       0 1572684738
901004511.787419  +l 2734968798         78   12686240 1572684898   12686240       0      0       0 1572684738
901004511.807221  +l 2734968958         79   12686400 1572684898   12686400       0      0       0 1572684738
901004511.827215  +l 2734969118         80   12686560 1572684898   12686560       0      0       0 1572684738
901004511.847214  +l 2734969278         81   12686720 1572684898   12686720       0      0       0 1572684738
901004511.867213  +l 2734969438         82   12686880 1572684898   12686880       0      0       0 1572684738
```

APPENDIX 6:

Matlab implementation of algorithm 1

```
load rx1 -ascii;               % received headers
rx_ts = rx1(:,1);                  % timestamps of received packets
rx_packets = size(rx_ts)       % number of packets received

load tx1 -ascii;               % transmitted headers
tx_sd = tx1(:,2);                  % silence detector values
tx_ts = tx1(:,1);                  % timestamps of sent packets

tx_packets = 0;
for j = 1:size(tx_sd)          % count nr. of transmitted packets
            if tx_sd(j)==1         % part of talkspurt if=1
            tx_packets = tx_packets+1;
            end
end

tx_packets % if tx_packets=rx_packets, no packets lost in network

j = 1; k = 1; l = tx_packets;

% compute 2-dim. matrix from transmitted packets
% 1.col=timestamp, 2.col indicates beginning of talkspurt

while (l > 0)
            while (tx_sd(j) ~= 1)
                j = j+1;    % find beginning of talkspurt
            end
                           % compensate clock skew
        sent_packets(k,1) = tx_ts(j-1)+0.0000013*(j-1);
        sent_packets(k,2) = 1; % mark beg. of talkspurt with 1
        j = j+2;
        k = k+1;
        l = l-1;

        while (tx_sd(j) ~= 0)
                j = j+1;
                if (tx_sd(j) == 1)
                        sent_packets(k,1) = tx_ts(j-1)+0.0000013*(j-1);
                        sent_packets(k,2) = 0; % mark rest of talkspurt with 0
                        k = k+1;
                        l = l-1;
                end
        end
end

% compute delay vector

for i = 1:tx_packets
        delay(i) = rx_ts(i) - sent_packets(i,1);
end

% calculate minimum delay

delay_min = delay(1);
for i = 2:tx_packets
            if (delay(i) < delay_min)
```

```
                delay_min = delay(i);
        end
end

delay_min

% subtract min. delay i.e. clock difference and propagation delay

for i = 1:tx_packets
        delay(i) = delay(i)-delay_min;
end

% plot(delay)

x = 1;
for beta = 0.5:0.5:20
beta

% calculate playout delays

alpha = 0.999;
delay_k = delay(1);
var_k = abs(delay(2)-delay(1));
pl_delay(1) = delay_k+beta*var_k;

for i = 2:tx_packets
      delay_k_minus1 = delay_k;
      delay_k = alpha * delay_k_minus1 + (1-alpha) * delay(i);
      var_k_minus1 = var_k;
      var_k = alpha * var_k_minus1 + (1-alpha) * ( abs(delay_k_minus1-delay(i)) );
      pl_delay(i) = delay_k + beta * var_k;
end

% plot(pl_delay)

k = 2; collisions = 0; pl_del = pl_delay(1);

% playout delay must be constant during a talkspurt

while (k < tx_packets)
      if (sent_packets(k,2) == 1)       % check that talkspurts do not overlap
                if ( (sent_packets(k,1) + pl_delay(k)) < (sent_packets(k-1,1) + pl_delay(k-1)
+ 0.02) )
                  collisions = collisions + 1;
                  pl_delay(k) = sent_packets(k-1,1)+pl_delay(k-1)+0.02-sent_packets(k,1);
            end
            pl_del = pl_delay(k);
            k = k+1;
      end
      while (sent_packets(k,2) == 0)
            pl_delay(k) = pl_del;
            if (k == tx_packets)
                  break;
            else
                  k = k+1;
            end
      end
end
```

```
collisions

% plot(pl_delay)

late = 0;

% count late arrived packets

for i = 1:tx_packets
        if ( rx_ts(i) > ( sent_packets(i,1) + delay_min + pl_delay(i) ) )
                late = late + 1;
        end
end

% compute packet loss rate

loss = (late/tx_packets)*100

% compute average playout delay

sum = 0;
for i = 1:tx_packets
        sum = sum + pl_delay(i);
end

delay_mean = sum/tx_packets

res(x,1) = loss;
res(x,2) = delay_mean;
res(x,3) = collisions;
x = x+1;
end

save t1a1_999 res -ascii;
plot(res(:,1),res(:,2))
```

APPENDIX 7:

Matlab implementation of algorithm 2

```
load rx1 -ascii;              % received headers
rx_ts = rx1(:,1);                % timestamps of received packets
rx_packets = size(rx_ts)       % number of packets received

load tx1 -ascii;              % transmitted headers
tx_sd = tx1(:,2);                % silence detector values
tx_ts = tx1(:,1);                % timestamps of sent packets

tx_packets = 0;
for j = 1:size(tx_sd)         % count nr. of transmitted packets
        if tx_sd(j)==1        % part of talkspurt if=1
        tx_packets = tx_packets+1;
        end
end

tx_packets % if tx_packets=rx_packets, no packets lost in network

j = 1; k = 1; l = tx_packets;

% compute 2-dim. matrix from transmitted packets
% 1.col=timestamp, 2.col indicates beginning of talkspurt

while (l > 0)
        while (tx_sd(j) ~= 1)
            j = j+1;    % find beginning of talkspurt
        end
                        % compensate clock skew
     sent_packets(k,1) = tx_ts(j-1)+0.0000013*(j-1);
     sent_packets(k,2) = 1; % mark beg. of talkspurt with 1
     j = j+2;
     k = k+1;
     l = l-1;

     while (tx_sd(j) ~= 0)
          j = j+1;
          if (tx_sd(j) == 1)
               sent_packets(k,1) = tx_ts(j-1)+0.0000013*(j-1);
               sent_packets(k,2) = 0; % mark rest of talkspurt with 0
               k = k+1;
               l = l-1;
          end
     end
end

% compute delay vector

for i = 1:tx_packets
     delay(i) = rx_ts(i) - sent_packets(i,1);
end

% calculate minimum delay

delay_min = delay(1);
for i = 2:tx_packets
        if (delay(i) < delay_min)
```

```
                delay_min = delay(i);
        end
end

delay_min

% subtract min. delay i.e. clock difference and propagation delay

for i = 1:tx_packets
        delay(i) = delay(i)-delay_min;
end

% plot(delay)

x = 1;
for beta = 0.5:0.5:20
beta

% calculate playout delays

alpha = 0.990;
mode = 1;           % 1=NORMAL, 2=SPIKE
spikes = 0;
sp_var = 0;
delay_k = delay(1);
var_k = abs(delay(2)-delay(1));
pl_delay(1) = delay_k+beta*var_k;

for i = 2:tx_packets
     delay_k_minus1 = delay_k;
     var_k_minus1 = var_k;

     if (abs(delay(i)-delay(i-1)) > (var_k_minus1*2+0.8))  % mode = SPIKE
          mode = 2;
               spikes = spikes + 1;
          sp_var = 0;
     else                                    % mode = NORMAL
          if (i==2)                          % if i = 2 can't subtract delay(0)
               sp_var = sp_var/2 + abs((delay(i) - delay(i-1))/4);
          else
               sp_var = sp_var/2 + abs((2*delay(i) - delay(i-1) - delay(i-2))/8);
          end

          if (sp_var <= 0.063)
               mode = 1;
          end
     end

     if (mode == 1)
          delay_k = alpha*delay_k_minus1+(1-alpha)*delay(i); % NORMAL mode
     else
          delay_k = delay_k_minus1 + abs(delay(i) - delay(i-1));  % SPIKE mode
     end

     var_k = alpha*var_k_minus1+(1-alpha)*(abs(delay_k_minus1-delay(i)));
     pl_delay(i) = delay_k + beta * var_k;
end

spikes
```

```
% plot(pl_delay)

k = 2; collisions = 0; pl_del = pl_delay(1);

% playout delay must be constant during a talkspurt

while (k < tx_packets)
        if (sent_packets(k,2) == 1)        % check that talkspurts do not overlap
                if ( (sent_packets(k,1) + pl_delay(k)) < (sent_packets(k-1,1) + pl_delay(k-1) +
0.02) )
                            collisions = collisions + 1;
                            pl_delay(k) = sent_packets(k-1,1)+pl_delay(k-1)+0.02-sent_packets(k,1);
                end
                pl_del = pl_delay(k);
                k = k+1;
        end
        while (sent_packets(k,2) == 0)
                pl_delay(k) = pl_del;
                if (k == tx_packets)
                         break;
                else
                         k = k+1;
                end
        end
end

collisions

plot(pl_delay)

late = 0;

% count late arrived packets

for i = 1:tx_packets
        if ( rx_ts(i) > ( sent_packets(i,1) + delay_min + pl_delay(i) ) )
                  late = late + 1;
        end
end

% compute packet loss rate

loss = (late/tx_packets)*100

% compute average playout delay

sum = 0;
for i = 1:tx_packets
        sum = sum + pl_delay(i);
end

delay_mean = sum/tx_packets

res(x,1) = loss;
res(x,2) = delay_mean;
res(x,3) = spikes;
res(x,4) = collisions;
x = x+1;
end
```

```
 save t1a2_990 res -ascii;
plot(res(:,1),res(:,2))
```

APPENDIX 8:

Matlab implementation of algorithm 3

```
load rx1 -ascii;              % received headers
rx_ts = rx1(:,1);                % timestamps of received packets
rx_packets = size(rx_ts)       % number of packets received

load tx1 -ascii;              % transmitted headers
tx_sd = tx1(:,2);                % silence detector values
tx_ts = tx1(:,1);                % timestamps of sent packets

tx_packets = 0;
for j = 1:size(tx_sd)          % count nr. of transmitted packets
        if tx_sd(j)==1         % part of talkspurt if=1
        tx_packets = tx_packets+1;
        end
end

tx_packets % if tx_packets=rx_packets, no packets lost in network

j = 1; k = 1; l = tx_packets;

% compute 2-dim. matrix from transmitted packets
% 1.col=timestamp, 2.col indicates beginning of talkspurt

while (l > 0)
        while (tx_sd(j) ~= 1)
           j = j+1;    % find beginning of talkspurt
        end
                      % compensate clock skew
      sent_packets(k,1) = tx_ts(j-1)+0.0000013*(j-1);
      sent_packets(k,2) = 1; % mark beg. of talkspurt with 1
      j = j+2;
      k = k+1;
      l = l-1;

      while (tx_sd(j) ~= 0)
            j = j+1;
            if (tx_sd(j) == 1)
                  sent_packets(k,1) = tx_ts(j-1)+0.0000013*(j-1);
                  sent_packets(k,2) = 0; % mark rest of talkspurt with 0
                  k = k+1;
                  l = l-1;
            end
      end
end

% compute delay vector

for i = 1:tx_packets
      delay(i) = rx_ts(i) - sent_packets(i,1);
end

% calculate minimum delay

delay_min = delay(1);
for i = 2:tx_packets
        if (delay(i) < delay_min)
```

```
                delay_min = delay(i);
        end
end

delay_min

% subtract min. delay i.e. clock difference and propagation delay

for i = 1:tx_packets
        delay(i) = delay(i)-delay_min;
end

% compute max. delay

del_max = 0;
for i = 1:length(delay)
     if delay(i) > del_max
            del_max = delay(i);
     end
end

del_max
unit = 0.010;    % delay distribution step size
w = 5000;        % window size
head = 4;        % constant to detect beginning of spike
tail = 2;        % constant to detect the end of spike
mode = 1;        % normal mode

V = [0.0005 0.0005 0.001 0.003 0.005 0.01 0.01 0.01 0.02 0.02 0.02 0.02 0.02 0.03 0.03
0.05 0.05];

x = 1;
q = 1;

while (q >= 0.80)
q     % percentile of delay distribution

for i = 1:ceil(del_max/unit)   % initialize delay distr. vector to zero
      del_distr(i) = 0;
end

for i = 1:w        % initialize delay statistics vector
      del_stat(i) = delay(i);
      pl_delay(i) = 0;
end

for i = 1:w        % initialize delay distr. vector
      if del_stat(i) == 0
             del_stat(i) = 0.001;
      end
      del_distr(ceil(del_stat(i)/unit)) = del_distr(ceil(del_stat(i)/unit)) + 1;
end

%plot(del_distr)

count = 0; curr_ind = 0;

while (count < w*q)      % initialize index to delay distr. vector
     count = count + del_distr(curr_ind + 1);
```

```
       curr_ind = curr_ind +1;
end

old_d = delay(w); spikes = 0;

for i = w:(length(delay)-1)

        if (mode == 2)
                if (delay(i+1) <= tail*old_d) % the end of spike
                        mode = 1;
                end
        else
                if (delay(i+1) > head*curr_ind*unit) % the beginning of spike
                        mode = 2;
                        old_d = curr_ind*unit;
                        spikes = spikes + 1;
                else

                     if (ceil(del_stat(rem(i,w)+1)/unit) <= curr_ind) % if delay inside distr.,
subtract from count
                        count = count-1;
                     end

                     % remove delay value that falls outside window

                     del_distr(ceil(del_stat(rem(i,w)+1)/unit)) =
del_distr(ceil(del_stat(rem(i,w)+1)/unit))-1;

                     if delay(i+1) == 0
                        delay(i+1) = 0.001;
                     end
                        del_stat(rem(i,w)+1) = delay(i+1);

                     % add new delay value to distribution

                     del_distr(ceil(del_stat(rem(i,w)+1)/unit)) =
del_distr(ceil(del_stat(rem(i,w)+1)/unit))+1;

                     if (ceil(del_stat(rem(i,w)+1)/unit) <= curr_ind)
                        count = count+1;
                     end

                     % count new index

                     while (count > w*q)
                        count = count-del_distr(curr_ind);
                        curr_ind = curr_ind-1;
                     end

                     while (count < w*q)
                        curr_ind = curr_ind+1;
                        count = count+del_distr(curr_ind);
                     end

                end
        end

        % compute playout delay
        if (mode == 2)
                pl_delay(i+1) = delay(i+1);
        else
                pl_delay(i+1) = curr_ind*unit;
        end
end

%plot(pl_delay)
spikes

k = w; collisions = 0; pl_del = pl_delay(w+1);

% playout delay must be constant during a talkspurt

while (k < tx_packets)
        if (sent_packets(k,2) == 1)        % check that talkspurts do not overlap
                        if ( (sent_packets(k,1) + pl_delay(k)) < (sent_packets(k-1,1) + pl_delay(k-1)
+ 0.02) )
                            collisions = collisions + 1;
                            pl_delay(k) = sent_packets(k-1,1)+pl_delay(k-1)+0.02-sent_packets(k,1);
                end
                pl_del = pl_delay(k);
                k = k+1;
        end
        while (sent_packets(k,2) == 0)
                pl_delay(k) = pl_del;
                if (k == tx_packets)
                        break;
                else
                        k = k+1;
                end
        end
end

collisions

%plot(pl_delay)

late = 0;

% count late arrived packets

for i = (w+1):tx_packets
        if ( rx_ts(i) > ( sent_packets(i,1) + delay_min + pl_delay(i) ) )
                 late = late + 1;
        end
end

% compute packet loss rate

loss = (late/(tx_packets-w))*100

% compute average playout delay

sum = 0;
for i = (w+1):tx_packets
        sum = sum + pl_delay(i);
end
```

```
delay_mean = sum/(tx_packets-w)

res(x,1) = loss;
res(x,2) = delay_mean;
res(x,3) = spikes;
res(x,4) = collisions;

q = q - V(x);
x = x+1;
end

save t1a3_5k res -ascii;
plot(res(:,1),res(:,2))
```

APPENDIX 9:

Matlab implementation of algorithm 4

```
load rx2 -ascii;              % received headers
rx_ts = rx2(:,1);                 % timestamps of received packets
rx_packets = size(rx_ts)       % number of packets received

load tx2 -ascii;              % transmitted headers
tx_sd = tx2(:,2);                 % silence detector values
tx_ts = tx2(:,1);                 % timestamps of sent packets

tx_packets = 0;
for j = 1:size(tx_sd)         % count nr. of transmitted packets
          if tx_sd(j)==1        % part of talkspurt if=1
          tx_packets = tx_packets+1;
          end
end

tx_packets % if tx_packets=rx_packets, no packets lost in network

j = 1; k = 1; l = tx_packets;

% compute 2-dim. matrix from transmitted packets
% 1.col=timestamp, 2.col indicates beginning of talkspurt

while (l > 0)
          while (tx_sd(j) ~= 1)
              j = j+1;    % find beginning of talkspurt
          end
                        % compensate clock skew
      sent_packets(k,1) = tx_ts(j-1)+0.0000013*(j-1);
      sent_packets(k,2) = 1; % mark beg. of talkspurt with 1
      j = j+2;
      k = k+1;
      l = l-1;

      while (tx_sd(j) ~= 0)
            j = j+1;
            if (tx_sd(j) == 1)
                  sent_packets(k,1) = tx_ts(j-1)+0.0000013*(j-1);
                  sent_packets(k,2) = 0; % mark rest of talkspurt with 0
                  k = k+1;
                  l = l-1;
            end
      end
end

% compute delay vector

for i = 1:tx_packets
      delay(i) = rx_ts(i) - sent_packets(i,1);
end

% calculate minimum delay

delay_min = delay(1);
for i = 2:tx_packets
          if (delay(i) < delay_min)
```

```
                delay_min = delay(i);
        end
end

delay_min

% subtract min. delay i.e. clock difference and propagation delay

for i = 1:tx_packets
        delay(i) = delay(i)-delay_min;
end

% compute max. delay

del_max = 0;
for i = 1:length(delay)
     if delay(i) > del_max
           del_max = delay(i);
     end
end

del_max
unit = 0.010;     % delay distribution step size
w = 5000;         % window size
head = 4;         % constant to detect beginning of spike
tail = 2;         % constant to detect the end of spike
mode = 1;         % normal mode

V = [0.0005 0.0005 0.001 0.003 0.005 0.01 0.01 0.01 0.02 0.02 0.02 0.02 0.02 0.03 0.03
0.05 0.05];
W = [0.5 0.5 0.25 0.5 1 1.25 1.25 1 1 0.75 0.75 0.5 0.5 0.25 0.1 0.1];

x = 1;
q = 1;
beta = 10.5;

while (q >= 0.80)
q      % percentile of delay distribution
beta

for i = 1:ceil(del_max/unit)   % initialize delay distr. vector to zero
      del_distr(i) = 0;
end

for i = 1:w         % initialize delay statistics vector
      del_stat(i) = delay(i);
      pl_delay(i) = 0;
end

for i = 1:w         % initialize delay distr. vector
      if del_stat(i) == 0
            del_stat(i) = 0.001;
      end
      del_distr(ceil(del_stat(i)/unit)) = del_distr(ceil(del_stat(i)/unit)) + 1;
end

%plot(del_distr)

count = 0; curr_ind = 0;
```

```
while (count < w*q)       % initialize index to delay distr. vector
        count = count + del_distr(curr_ind + 1);
        curr_ind = curr_ind +1;
end

% calculate playout delays

alpha = 0.998002;
delay_k = delay(1);
var_k = abs(delay(2)-delay(1));
pl_delay(1) = delay_k+beta*var_k;
old_d = delay(1); spikes = 0;

for i = 1:(length(delay)-1)

         if (mode == 2)
                  if (delay(i+1) <= tail*old_d) % the end of spike
                        mode = 1;
                  end
         else
                  if (delay(i+1) > head*curr_ind*unit) % the beginning of spike
                        mode = 2;
                        old_d = curr_ind*unit;
                        spikes = spikes + 1;
                  else

                        if (i<w)
                              delay_k_minus1 = delay_k;
                              delay_k = alpha * delay_k_minus1 + (1-alpha) * delay(i+1);
                              var_k_minus1 = var_k;
                              var_k = alpha * var_k_minus1 + (1-alpha) * ( abs(delay_k_minus1-
delay(i+1)) );
                        else

                              if (ceil(del_stat(rem(i,w)+1)/unit) <= curr_ind) % if delay inside distr.,
subtract from count
                                    count = count-1;
                              end

                              % remove delay value that falls outside window

                              del_distr(ceil(del_stat(rem(i,w)+1)/unit)) =
del_distr(ceil(del_stat(rem(i,w)+1)/unit))-1;

                              if delay(i+1) == 0
                                    delay(i+1) = 0.001;
                              end
                                  del_stat(rem(i,w)+1) = delay(i+1);

                              % add new delay value to distribution

                              del_distr(ceil(del_stat(rem(i,w)+1)/unit)) =
del_distr(ceil(del_stat(rem(i,w)+1)/unit))+1;

                              if (ceil(del_stat(rem(i,w)+1)/unit) <= curr_ind)
                                    count = count+1;
                              end

                              % count new index
```

```
                while (count > w*q)
                    count = count-del_distr(curr_ind);
                    curr_ind = curr_ind-1;
                end

                while (count < w*q)
                    curr_ind = curr_ind+1;
                    count = count+del_distr(curr_ind);
                end

            end
        end
    end

    % compute playout delay

    if (mode == 2)
        pl_delay(i+1) = delay(i+1);
    else
        if (i<w)
            pl_delay(i+1) = delay_k + beta * var_k;
        else
            pl_delay(i+1) = curr_ind*unit;
        end
    end
end

%plot(pl_delay)
spikes

k = 2; collisions = 0; pl_del = pl_delay(1);

% playout delay must be constant during a talkspurt

while (k < tx_packets)
    if (sent_packets(k,2) == 1)        % check that talkspurts do not overlap
            if ( (sent_packets(k,1) + pl_delay(k)) < (sent_packets(k-1,1) + pl_delay(k-1)
+ 0.02) )
                collisions = collisions + 1;
                pl_delay(k) = sent_packets(k-1,1)+pl_delay(k-1)+0.02-sent_packets(k,1);
        end
        pl_del = pl_delay(k);
        k = k+1;
    end
    while (sent_packets(k,2) == 0)
        pl_delay(k) = pl_del;
        if (k == tx_packets)
            break;
        else
            k = k+1;
        end
    end
end

collisions

%plot(pl_delay)

late = 0;
```

```
% count late arrived packets

for i = 1:tx_packets
        if ( rx_ts(i) > ( sent_packets(i,1) + delay_min + pl_delay(i) ) )
                late = late + 1;
        end
end

% compute packet loss rate

loss = (late/tx_packets)*100

% compute average playout delay

sum = 0;
for i = 1:tx_packets
        sum = sum + pl_delay(i);
end

delay_mean = sum/tx_packets

res(x,1) = loss;
res(x,2) = delay_mean;
res(x,3) = spikes;
res(x,4) = collisions;

q = q - V(x);
beta = beta - W(x);
x = x+1;
end

save t2a4 res -ascii;
plot(res(:,1),res(:,2))
set(gca,'XScale','log');
```