Adding NTP and RTCP to a SIP User Agent

FRANZ MAYER

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2006

COS/CCS 2006-14

Adding NTP and RTCP to a SIP User Agent

Franz Mayer

In Partial Fulfillment

of the Requirements for the

Master of Science in

Information Systems and Management

Advisor and Examiner: G.Q. Maguire Jr.

Department of Communication Systems
Royal Institute of Technology (Kungliga Tekniska Högskolan, KTH)

Stockholm, June 19th 2006

Abstract

With its enormous potential Voice over Internet Protocol is one of the latest buzzwords in information technology. Despite the numerous advantages of Voice over IP, it is a major technical challenge to achieve a similar call quality as experienced in the ordinary Public Switched Telephone Network.

This thesis introduces standardized Internet protocols for Voice over IP, such as Session Initiation Protocol (SIP), Real-time Transport Protocol (RTP), in its background chapter. In order to provide better Quality of Service (QoS) Voice over IP applications should support a feedback mechanism, such as the Real-time Control Protocol (RTCP), and use accurate timing information, provided by the Network Time Protocol (NTP). Additionally this thesis considers synchronization issues in calls with two and more peers.

After a rather academic overview of Voice over IP, the open source real-time application “minisip”, a SIP user agent, and its operation and structure for handling audio streams will be introduced. Minisip was extended by an implementation of NTP and RTCP to provide a test platform for this thesis.

A clear conclusion is that the addition of global time helps facilitate synchronization of multiple streams from clients located any where in the network and in addition the ability to make one-way delay measurements helps SIP user agents to provide better quality audio to their users.

Sammanfattning

Röst över IP, eller Internettelefoni baserad på “Voice over Internet Protocol” (VoIP), har med sin stora potential blivit ett av de senaste modeorden inom informationsteknologin. Vid sedan av ett antal fördelar med VoIP så innebär det en stor teknisk utmaning att uppnå en likadan samtalskvalitet som i det vanliga, fasta, telenätet.

I den här uppsatsen beskrivs hur tjänstevalitet för VoIP kan förbättras genom att noggrant tidssynkronisera de (två eller flera) klienter som deltar i ett telefonsamtal. För detta krävs dels en återkopplingsmekanism, såsom “Real-time Control Protocol” (RTCP), samt en gemensam tidsuppfattning i de inblandade klienterna, vilket kan uppnås med hjälp av “Network Time Protocol” (NTP). Dessa protokoll, liksom de övriga Internet-standarder som VoIP baseras på (såsom “Session Initiation Protocol” (SIP) och “Real-time Transport Protocol” (RTP), beskrivs inledningsvis i uppsatsen.

För studien har en SIP-klient baserad på öppen källkod använts (“Minisip”), och utökats med NTP och RCTP funktionalitet för att testa den föreslagna förbättringen av VoIP. En tydlig slutsats är att kännedom om en “global tid” möjliggör synkronisering av multipla ljudströmmar från klienter som befinner sig på olika nätverk. Möjligheten att mäta paketfördröjningen (envägs) bidrar också till en förbättrad ljudkvalitet.

Table of Contents

List of Figures

List of Tables

List of Terms

CODEC	Coder / Decoder
CSRC	Contributing Source Identifier(s)
DHCP	Dynamic Host Configuration Protocol
DLRR	Delay since Last Receiver Report
DLSR	Delay since Last Sender Report
GPL	GNU General Public Licence
GPS	Global Positioning System
GUI	Graphical User Interface
Hz	Hertz
IM	Instant Messaging
IP	Internet Protocol
LSR	Last Sender Report
LSW	Least Significant Word
ms	Millisecond
µs	Microsecond
MSW	Most Significant Word
NIST	U.S. Nation Institute of Standards and Technology
ns	Nanosecond
NTP	Network Time Protocol
OS	Operating System
PPM	Part Per Million
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RR	Receiver Report
RTC	Real-Time Clock
RTCP	Real-time Control Protocol
RTCP XR	RTP Control Protocol Extended Reports
(S)RTP	(Secure) Real-time Transport Protocol
RTT	Round Trip (Delay) Time
SDES	Source Description
SIP	Session Initiation Protocol
SR	Sender Report
SSRC	Synchronization Source Identifier

SVN	Subversion
TCP	Transmission Control Protocol
TTL	Time To Live
UA	User Agent
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
UTC	Universal Time Coordinated
VoIP	Voice over Internet Protocol
WLAN	Wireless Local Area Network
XR	Extended Reports

1 Introduction

Internet telephony - also known as “Voice over Internet Protocol”, “Voice over IP” or “VoIP” - is becoming more and more popular. VoIP is easy to use and can used even without a computer, since various companies, such as Cisco Systems, offers VoIP telephones. Many consumers and companies are switching now from the Public Switched Telephone Network (PSTN) to VoIP, because it offers a stable, secure, and low cost way to communicate. Besides, it provides additional services, such as Instant Messaging (IM), video conferencing, sending files, calling landline numbers at affordable rates, and much more. Therefore the market for VoIP products and its potential for users is growing enormously.

However, providing a continious flow of voice through a network, such as the Internet, is the main challenge of VoIP. For an interactive dialogue it is important that the speech is in order and arrives on time, so that participants can understand each other and can react (i.e. answer appropriately). To achieve a similar quality to the ordinary PSTN sufficient bandwidth, correct timing, and a feedback mechanism are needed.

Without an internet connection to transfer audio data through the Internet, Voice over IP can not exist. However, even a very large bandwidth does not guarantee that audio packets are played in time at the receiving peer's speaker (or headphones). When other applications use most of the bandwidth, there might not be enough bandwidth for transmitting the session's audio packets, thus the audio-quality might suffer. To check if the packets are received in sufficient time a feedback mechanism is necessary to ensure Quality of Service (QoS).

When communicating with more than one peer, as in conference calls, timing of packets is important. The standard VoIP audio protocol, Real-Time Protocol (RTP, see Section 2.3), only provides relative timestamps, i.e. the timing is not done with respect to global time. Therefore clocks of all the participants should be synchronized to provide accurate RTP timestamps; this can be achieved by using the Network Time Protocol (NTP, see Section 2.5). In Section 2.4 we will take a closer look at the synchronization problem in VoIP calls.

The Real-Time Control Protocol (RTCP, see Section 2.6) provide QoS feedback for the RTP traffic to each RTP receiver. RTCP requires NTP timestamps to relate events to global time; this can then be used to calculate network delay, for example for using different CODECs [21], re-sampling (via dynamic time wraping), or re-ordering audio packets.

1.1 Organization of this report

Chapter 2 provides the necessary background information concerning Voice over IP and its underlying protocols, e.g., Session Initiation Protocol (SIP, Section 2.2), Real-time Transport Protocol (RTP, Section 2.3), Network Time Protocol (NTP, Section 2.5), RTP Control Protocol (RTCP, Section 2.6), and RTP Control Protocol Extended Reports (RTCP XR, Section 2.7). Furthermore Section 2.4 considers intrastream and interstream synchronization.

Chapter 3 introduces the real-time application minisip. Section 3.1 describes how minisip can be used on top of Microsoft's Windows XP. The structure and operations of minisip, especially those for media streams, is described in Section 3.2. Measuring packet loss and checking sequence ordering will be discussed in Section 3.3. Finally, Section 3.4 presents the implementation of NTP in minisip.

Chapter 4 tests the code developed in the course of this thesis. Section 4.1 describes the test setup used for verifying the RTCP implementation, while Section 4.2 evaluates the resulting performance via these tests. Section 4.3 concerns further tests, for example for conference calls and video calls.

Chapter 5 presents some conclusions and describes future work building upon this thesis.

2 Background: Voice over IP

This chapter introduces the main standards for Voice over Internet Protocol (VoIP) for use by real-time applications. The general structure of VoIP applications, such as minisip, and its subsystems are depicted in Figure 2.1:

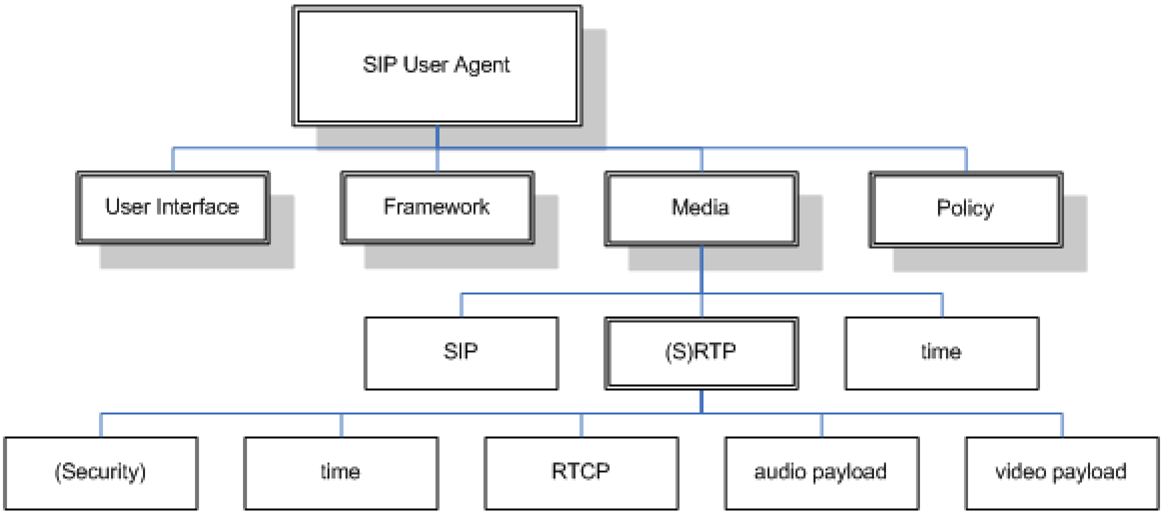

Figure 2.1: Hierarchical structure of an SIP application , such as minisip

This thesis mainly focuses on the Media subsystem, which handles all audio and video processing. The User Interface (UI), Policy, and Framework[1] subsystems are described in more detailed in [5]. Furthermore security will not be discussed, as this has been extensively described in numerous theses [12].

First of all, minisip will be briefly introduced (section 2.1) - for a deeper analysis see chapter 3. In section 2.2, the Session Initiation Protocol (SIP) is described. The core protocol for VoIP media streams, the Real-time Transport Protocol (RTP) is described in section 2.3. In section 2.4 synchronization issues in VoIP are discussed, which leads to the introduction of the Network Time Protocol (NTP) in Section 2.5. In sections 2.6 and 2.7 a number of protocols to support Quality of Service (QoS) and transfer of additional information are described.

2.1 The minisip user agent

minisip[2] is a SIP soft phone application developed by the Telecommunication Systems Lab in cooperation with the Center for Wireless Systems (Wireless@KTH) at the Royal Institute of Technology (“Kungliga Tekniska Högskolan”, KTH) in Kista, Sweden. The application is available as open source under the terms of a GNU General Public Licence (GPL) [12].

minisip provides a way to communicate in a secure way with other internet users based on Session Initiation Protocol (SIP). Alternative applications, such as Skype[3], include additional services, i.e. instant messaging and a highly developed graphical user interface (GUI).

---

1 In some minisip documents, such as [5], the framework subsystem is called “inter-subsystem communication”.

2 For more information see http://www.minisip.org/.

3 Skype™ is a Skype Technologies S.A. product. For more information see http://www.skype.com/

2.2 Session Initiation Protocol – SIP

There are a growing number of applications that require an exchange of data between session participants. The *Session Initiation Protocol* (SIP) defines a means for creation, modification, and termination of such sessions involving two or more participants. The session initiated by SIP can provide applications with real-time multimedia, i.e. enabling multiple Internet endpoints (*user agents*) to locate prospective session participants and to create sessions. SIP supports registration, invitation to sessions, and other requests [15].

SIP functionality has five different facets:

1. *User location* determines the location of an end point using an Uniform Resource Identifier (URI) - called a SIP URI.
2. *User availability* determines the willingness of the called party (user agent) to receive a call.
3. *User capabilities* determines selection of media and media parameters.
4. *Session setup* establishes session parameters for both called and calling party.
5. *Session management* provides for the transfer and termination of sessions, modification of session parameters, invocation of services, etc.

Figure 2.2 shows the basic functions of SIP: Consider that somebody wants to place a call to the cisco1 SIP phone physically located in the wireless lab using a minisip client, the calling user agent (in this case minisip) has to determine the location of the remote end point (facet 1).

To determine the callee's location *minisip* asks a SIP proxy (in this case *iptel.org*) to find the location of the SIP phone named *cisco1* (based upon its SIP URI – sip: cisco1@130.237.15.222). We see that. *minisip* (UA1) sends an INVITE request to its proxy (in F1), which forwards the invitation to the remote proxy (in F2). The proxy of minisip (*iptel.org*) returns the state of enquiry (Trying in F3) and the remote proxy (*kth.org*) contacts the remote user agent (UA2) indicating that there is an incoming call from minisip (with F4). When UA2 has been located, then the SIP phone will start ringing (step F6) and this information will be transmitted to minisip (in messages F7, F8).

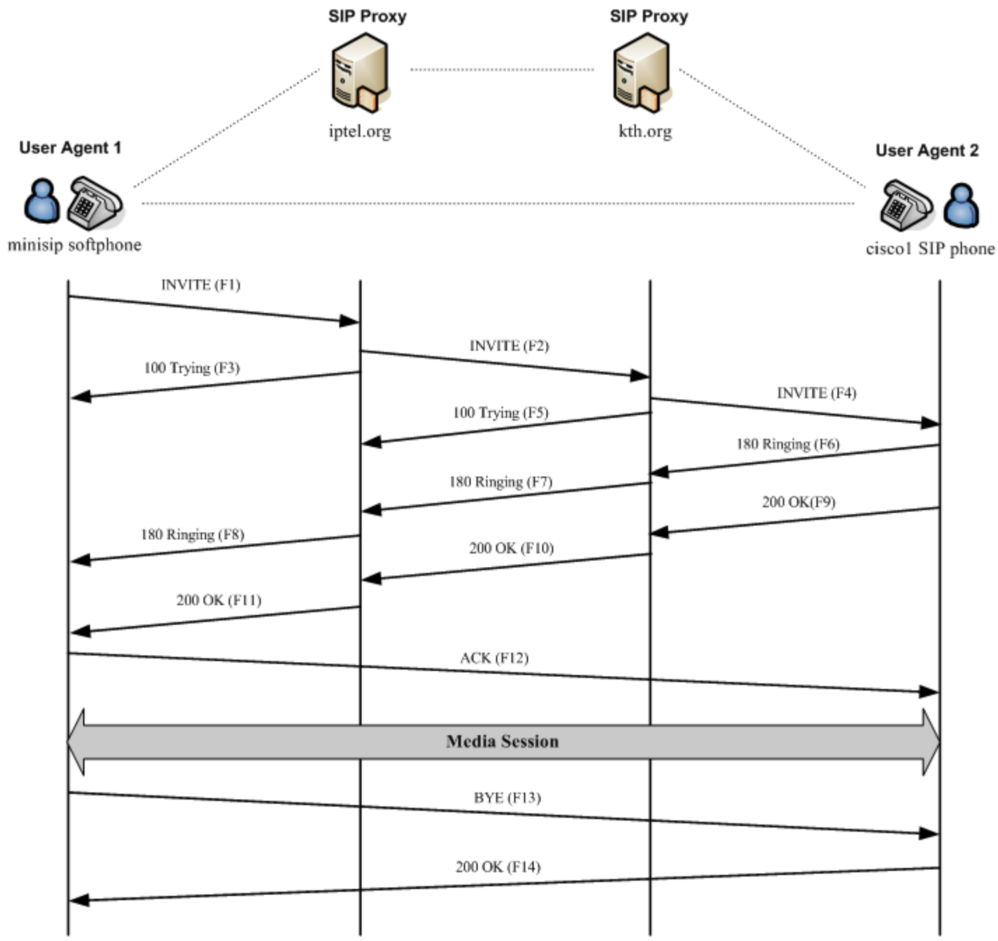

Figure 2.2: SIP session setup example with SIP trapezoid; based on [15]

If UA2 is willed to accept the call (facet 2), it will send a message (OK in F9), which will be transmitted through the proxies of both parties (see F10 & F11). UA1 will send an acknowledge message to UA2 (F12), and from this point on the two user agents will exchange the mutually agreed media data directly (i.e., peer-to-peer) as a media session.

When describing the operation of SIP, the SIP trapezoid is often mentioned. This means that only the SIP signalling messages (steps 1..14 mentioned above) are transferred through proxies, but all session media will be transferred directly between the peers.

2.3 Real-time Transport Protocol – RTP

The Real-time Transport Protocol (RTP) defines a standardized packet format for end-to-end network transport of media and usually utilizes the User Datagram Protocol (UDP) as its transport protocol. It is widely used for media applications such as interactive audio and video. RTP itself does not provide a defined Quality of Service (QoS) or guarantee timely delivery of packets. It does, however, provide the necessary timing and packet sequencing information to enable an ordered and continuous real-time data stream. RTP is further separated into the two components: RTP and Real-time Control Protocol (RTCP) [1], [17]. A basic introduction to these protocols can be found in [1], whereas [17] is the full specification for RTP.

Table 2.1: RTP fixed header fields [17]

0				8		16 31
V=2	P	X	CC	M	PT	sequence number
timestamp						
synchronization source (SSRC) identifier						
contributing source (CSRC) identifier(s) ...						

V (2 bits)	version, identifies the version of RTP, currently 2
P (1 bit)	padding bit
X (1 bit)	extension bit
CC (4 bits)	CSRC count, contains the number of CSRC identifiers that follow the fxed header
M (1 bit)	marker bit
PT (7 bits)	payload type, identifies the format of the RTP payload and determines its interpretation by the application

Table 2.1 shows the structure of a fixed RTP header. The 16 bit RTP *sequence number* is incremented by one for each RTP packet and should be initialized with a random (unpredictable) number to make it more robust against attacks. It can be used to detect packet loss and to restore packet sequence [17].

The 32 bit RTP *timestamp* specifies the sampling instant, which **must** be derived from a clock that increments monotonically and linearly. The clock needs a certain minimum resolution to calculate network delay variation (*jitter*) and to synchronize the RTP data stream. As with the RTP *sequence number* the initial value *should* be randomly chosen for security reasons. The RTP *timestamp* is then incremented by one for each sample, so when an audio application reads blocks containing 160 samples (e.g. a 20ms voice packet at a sampling rate of 8,000 Hz), the timestamp should be increased by 160 for each block[4].

The *synchronization source (SSRC) identifier* is a random 32-bit number, which is unique within a RTP session. There can be a list of up to 15 *contributing sources (CSRC) identifiers*, each of which is 32-bits long. Information about contributing sources is used to avoiding mixing a source in multiple times or sending a source its own traffic.

After the fixed header (Table 2.1) extensions and the RTP *payload*, the actual data field follows.

2.3.1 RTP Mixers and Translators

Not every participant of a conference has the same environment; they might differ in bandwidth, security, or be using another network protocol. Therefore two types of intermediaries are necessary to ensure portability and scalability: *mixers* and *translators*.

4 Note: while doing silence suppression the clock continues to increment, even though the sequence number will not increase – as RTP packets are not being sent, although samples continue to be made.

If a conference has participants with more limited bandwidth than other participants, a *mixer* should be installed prior to the low-bandwidth area. This *mixer* collects all incoming audio packets, then resynchronizes and transcodes the data into a lower-bandwidth packet stream. These packets are sent to lower-bandwidth participants, who may receive a reduced quality audio stream, but are still able to participate in the conference - despite their limited network connectivity. However, all other participants can send and receive the higher quality audio [17].

Many participants may use an application-level firewall that blocks IP packets. In this case, for each participant using a firewall, two *translators* are installed, one on each side of the firewall. The outside *translator* forwards all packets through a secure IP connection to the inside *translator*, which sends packets to all participants of the internal network. Additionally, translators can be used to translate the media data from one format to another - thus all the participants do not need to use the same coder/decoder (CODEC).

2.4 Synchronization in Voice over IP

For Voice over IP (VoIP), synchronization of media streams is essential to real-time communication. There are basically two different kinds of synchronization: *intrastream* and *interstream synchronization* [6].

2.4.1 Intrastream Synchronization

The goal of intrastream synchronization, also known as *playout scheduling*, is to play an ordered and continous media stream. It depends on hiding both packet loss and end-to-end delay variations (jitter) [6]. Figure 2.3 illustrates the problem of intrastream synchronization.

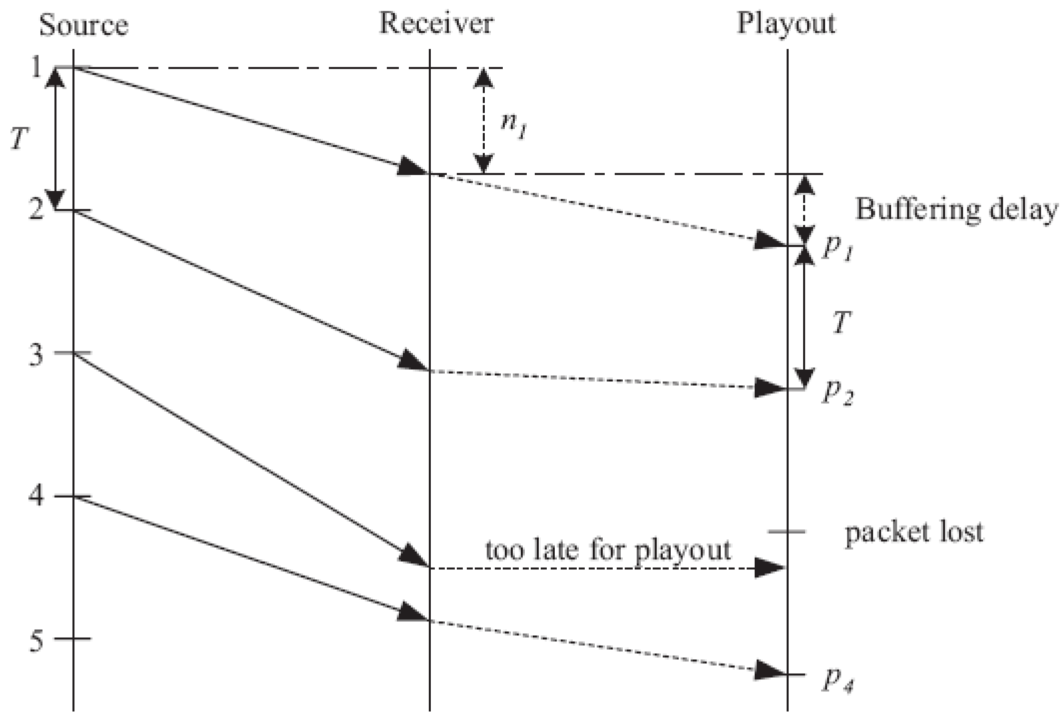

Figure 2.3: Playout scheduling problem [6], where
T is time to the next packet (also know as the sampling period),
n_1 is the network delay, and
p_n is when the packet is to be played.

There are various algorithms for intrastream synchronization most relying on timing information in form of timestamps. As described in section 2.3, RTP provides information on packet ordering (sequence number) and a relative delay (RTP timestamp).

2.4.2 Interstream Synchronization

While intrastream synchronization is packet orientated, interstream synchronization attempts to preserve the temporal relationship of two or more streams. For voice calls it has been shown that synchronization errors less than ±120ms are generally not noticeable [6]. Basically there are four reasons for synchronization errors:

- Clock skew
- Different initial collection times
- Different initial playback times
- Network delay variation (i.e., jitter)

Clock skew

When clocks tick at different speeds the result will be a gradual shift in synchronization. In Figure 2.4, for example, a sender sends packets while sampling at 8,000 Hertz (Hz), but receiver 1's clock ticks slightly slower. Therefore the playout of the received samples takes longer, which can lead to buffer overflow, because the sender produces more samples per unit time than receiver 1 is able to play in the same amount of absolute time. Whereas if the clock of receiver 2 is faster, the result will be a lack of samples to play. Usually clock skew is negligible compared to network delay, but may still be significant and thus require resampling of the samples before playout. When for example a clock deviates ± 0.5% from the original clock (e.g. sender's clock), slower clocks will play the sample at a sample rate of 8,040 of the original one and faster ones at a rate of 7,960 samples. Therefore every 200th packet, approximately every 4th second, the real-time application will either miss one packet (receiver 2) or have one packet too much in the buffer (receiver 1).

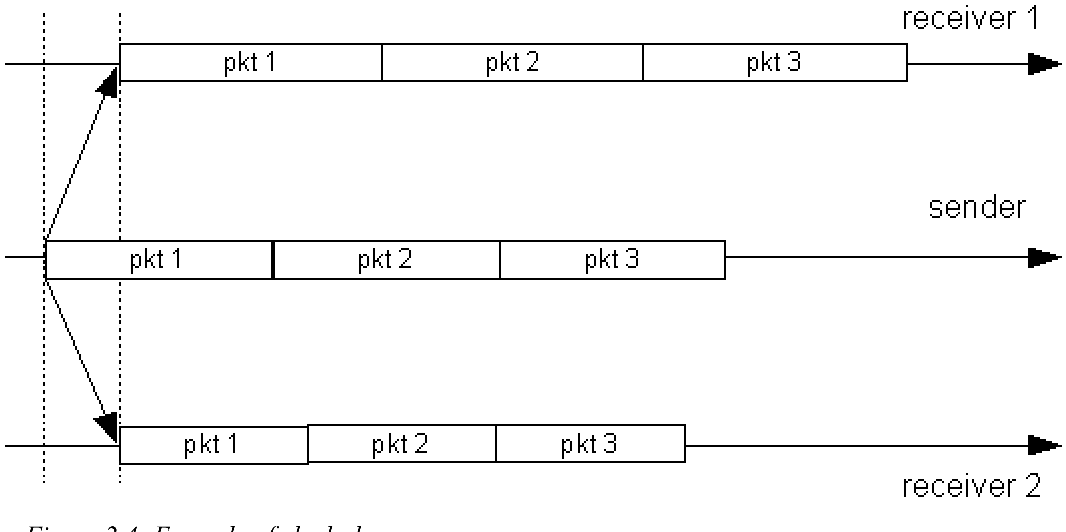

Figure 2.4: Example of clock skew

Different initial collection times

In distributed media sessions, synchronization is often required in conference calls or video games, for example. However, if the clocks are not synchronized, then the time when the transmission begins can differ; such as s_1 and s_2 in Figure 2.5.

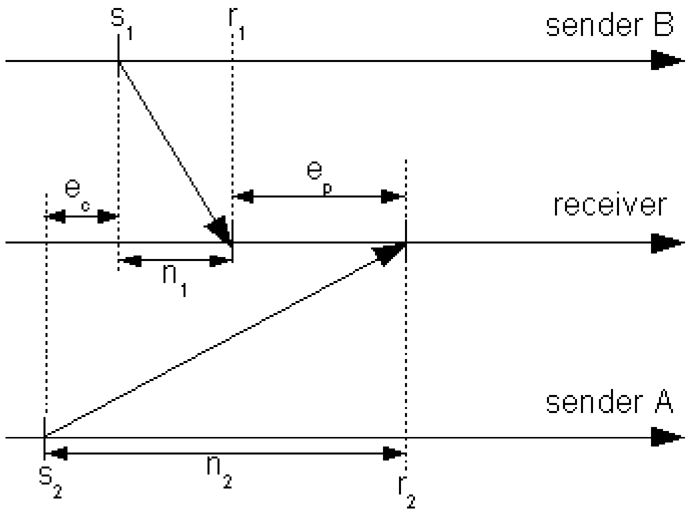

Figure 2.5: Interstream synchronization errors; based on [6], where
s_n is the time when sending an audio packet,
r_n is the time when the audio packet is received,
e_c is the collection error, and
e_p is the playout error.

Different initial playback times

In a conference every participant should hear the sample at the same time in order to have a continous discussion, which is perceived by all listeners as the same. Unfortunately, the playback or playout time can differ at receiver's side, shown as r_1 and r_2 in Figure 2.5 due to different network delays, jitter, or packetization (shown as collection delay in Figure 2.5) This so-called group synchronization can generally be achieved through a feedback meachanism, such as the RTP Control Protocol (RTCP, see section 2.6) provides. Once all participants have sent their feedback, everyone can adjust their buffer to accommodate the largest amount of jitter. Using global time, for example via Network Time Protocol (NTP, see section 2.5), the absolute playout time could be determined and each packet played at a specific global time!

Network delay variation

Different media streams take different routes, which can lead to varying delays; shown in Figure 2.5 as n_1 and n_2, thus resulting in different delays.

2.5 Network Time Protocol – NTP

The specification of the Network Time Protocol (NTP) is currently available in Version 3 of RFC-1305 and has been created to provide a mechanism to synchronize and coordinate time distribution [11].

In order to allow nodes attached to the network to accurately learn what time it is requires either that each node have its own accurate source of time or to learn about the current time from others via the network. In this section we will consider the second method in detail and describe exactly how information concerning time is transmitted over the network and how it is interpreted by the receivers. However, first we need to introduce some important terms [11], [20]:

Resolution is the smallest possible increase of time allowed by your clock; NTP provides a resolution about 200 picoseconds (= 0.2 nanosecond).

Precision is the smallest possible increase of time that can be computed by a program.

Accuracy determines how close a certain clock is to an official time reference such as Universal Time Coordinated (UTC).

Stability of a clock is how well it can maintain a constant frequency. The frequency of the typical clock hardware, however, is **never exactly** correct. Even a slight frequency error of 0.0012% or 12 PPM (Part Per Million) would cause such a clock to be off by roughly one second per day [20].

Stratum is a classification of NTP servers and their time quality, which includes *Precision*, *Accuracy*, and *Stability*.

A *Reference Clock* is a clock with a very high accuracy, which is typically a very expensive atomic clock. The Global Positioning System (GPS) utilizes a notion of time derived from atomic clocks and broadcasts this timing information by modulating a very long pseudo-random sequence. Such reference clocks are referred in NTP as *stratum 0* since they provide the highest possible time quality.

2.5.1 NTP Data Format

NTP utilizes several different messages, all of these messages use the same header. The NTP Message Header is shown in Table 2.2.

Table 2.2: NTP Message Header [11]

<table>
<tr><td colspan="3">0</td><td>8</td><td>16</td><td>31</td></tr>
<tr><td>LI</td><td>VN</td><td>mode</td><td>stratum</td><td>poll</td><td>precision</td></tr>
<tr><td colspan="6">root delay (32)</td></tr>
<tr><td colspan="6">root dispersion (32)</td></tr>
<tr><td colspan="6">reference identifier (32)</td></tr>
<tr><td colspan="6">reference timestamp (64)</td></tr>
<tr><td colspan="6">originate timestamp (64)</td></tr>
<tr><td colspan="6">receive timestamp (64)</td></tr>
<tr><td colspan="6">transmit timestamp (64)</td></tr>
<tr><td colspan="6">authenticator (optional) (96)</td></tr>
</table>

All *NTP timestamps* are represented as a 64-bit unsigned fixed-point number with an implied fraction point between the two 32-bit halves. The first 32-bits are the integer part and represents the seconds starting from 0.00 o'clock in January the 1st 1900. The second 32-bits are the fraction part, which splits each second into the exact resolution of NTP (2^{-32} second ~ 0.2 ns).

LI (2 bits)	*Leap Indicator*, for an impending leap second, which should be inserted or deleted respectively
VN (3 bits)	*Version Number*, which is currently 3 for NTP
mode (3 bits)	indicating the broadcast *mode*
stratum (8 bits)	indicating the stratum-level of the NTP-server
Poll (8 bits)	indicates the Poll Interval
Precision (8 bits)	indicating the precision of the local clock
Root Delay	indicates the total round trip delay
Root Dispersion	indicates the maximum error relative to the primary reference source
Reference Clock Identifier	identifies the particular reference clock
Reference Timestamp	local time when the local clock was last updated
Originate Timestamp	local time when the peer sent the latest NTP message
Receive Timestamp	local time when the NTP message from the peer arrived
Transmit Timestamp	local time, at which the NTP message departed the sender
Authenticator	When the NTP authentication mechanism is implemented, this contains the authenticator information.

2.5.2 Time Synchronization with NTP

Synchronization of time is done through several packet exchanges, each a request and reply pair, as it can be seen in Figure 2.6 below. When a client sends a request to a NTP server, the client stores its own local time (*Originate Timestamp*) into the NTP packet. When the server receives the packet, it will store its own local time of reception (*Receive Timestamp*) into the packet, and puts its local time (*Transmit Timestamp*) just before the packet will be transmitted back to the client. When the client receives the packet it will compute its own local time once more to estimate the round trip time (delay) [20]. This procedure has to be done several time to estimate delays in network. This allows the local node to compute the local clock offset.

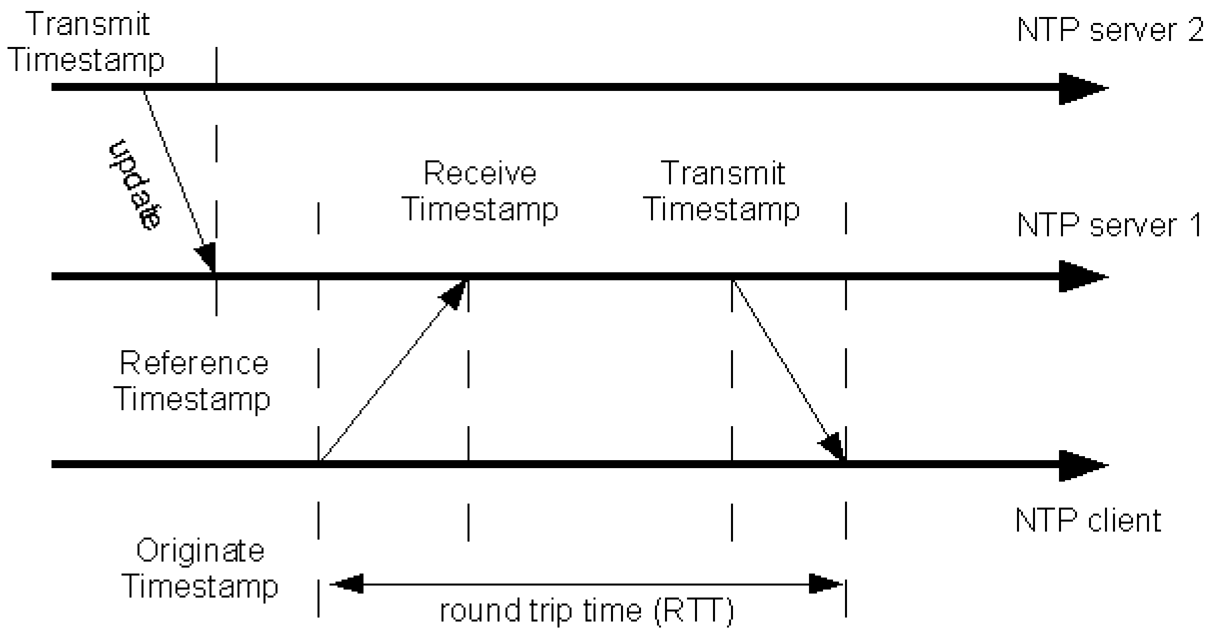

Figure 2.6: Timeline of NTP message exchange

There are different reasons and various terms for time differences between client and server. We define below terms following the terminology of [11], [20]:

Round Trip Delay Time (RTT) is the time required for the packet to be sent and for the response to be received. It can be defined as the time between when the request packet was sent and the when reply packet is received.

Offset is a time difference between two clocks.

Skew is the frequency difference between two clocks.

Clock Offset represents the amount by which to adjust the local clock to bring it into correspondence with the reference clock.

Dispersion is the maximum offset error (difference between local and reference clock).

2.5.3 Using NTP

Most operating systems (OS) - such as Microsoft's Windows[5], Linux derivates, etc. - supports NTP as a client and / or as a server. The structure of synchronizing NTP in an intranet might be as shown in Figure 2.7.

5 Windows™ is a Microsoft product. For more information see http://www.microsoft.com/windows/.

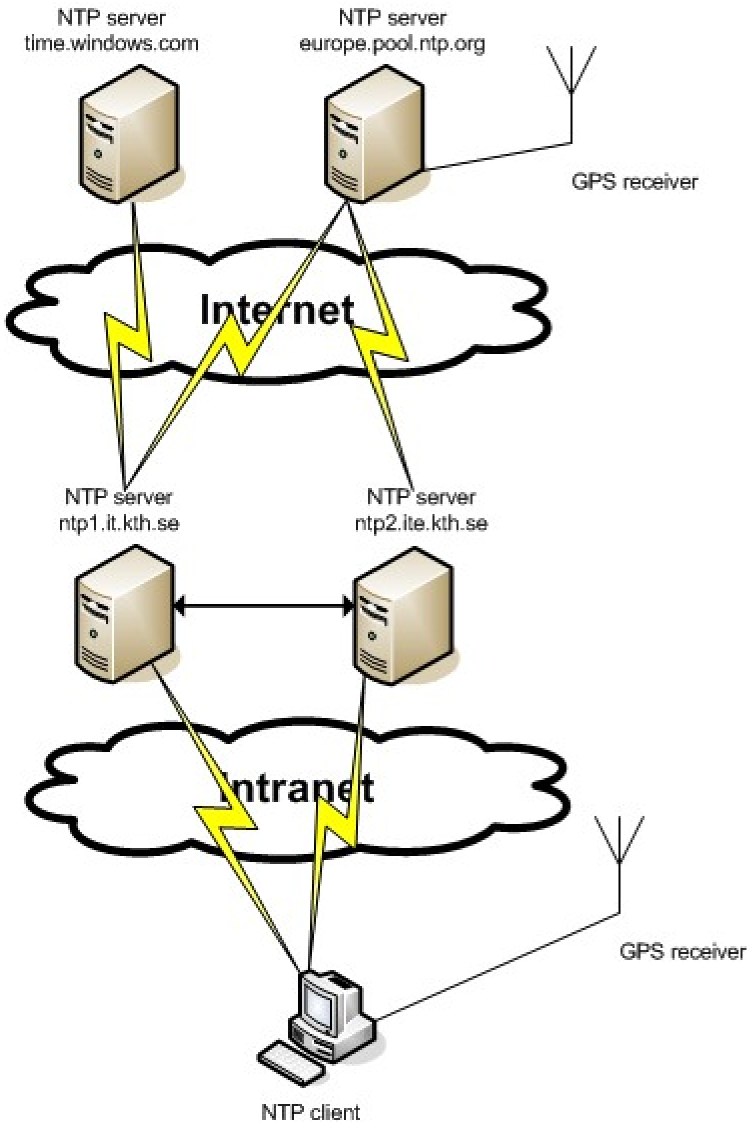

Figure 2.7: NTP synchronization

Microsoft's Windows versions XP and 2000 include the *Windows Time Service*[6], which supports NTP. To synchronize time, either double click on the Windows' clock (usually in the buttom right of the screen) and click "Update now" in the "Internet Time" tab or type the following Windows command[7]:

```
C:\>w32tm /resync /rediscover
```

To see the current offset of the local clock type the following command:

```
C:\>w32tm /monitor /computers:ntp1.kth.se
ntp1.kth.se [130.237.48.28]:
    ICMP: 1ms delay.
    NTP: +1.3863246s offset from local clock
        RefID: ntp1.sth.netnod.se [192.36.144.22]
```

After a short while it should say: "The time has been successfully synchronized with [...]". If it was not successful, it could be one of several reasons: First of all it could be that the firewall of the

6 See Control Panel > Administrative Tools > Services.

7 To start the Windows' command line interface click on start > Run ...

personal computer (PC) blocks UDP port 123, which NTP uses. This can be checked with the following Windows' command:

```
C:\>netstat -an |find "123"
```

If operating in a network - such as university or company network - it is likely that port 123 is blocked. This may be due to performance or security reasons. Using the internal NTP server saves network resources (i.e. traffic) and avoids having an open port; many peer-to-peer applications try to "tunnel out" past the firewall using standard application ports. If the port is blocked, then the internal NTP server has to be used, which can be found out either by asking the system administrator or by performing a network lookup[8]:

```
C:\>nslookup
Default Server:  res2.ns.kth.se
Address:  130.237.72.200

> search ntp1[9]
Server:  casio.ite.kth.se
Address:  130.237.48.28
Aliases:  ntp1.kth.se, ntp2.ite.kth.se
*** ntp1 can't find search: No response from server
```

The default NTP server[10] can be set as followed:

```
C:\>net time /setsntp:ntp1.kth.se,ntp2.ite.kth.se
```

To check parameters regarding NTP, type[11]:

```
C:\>w32tm /dumpreg /subkey:parameters
```

To automatically allow changes the Windows' Time service has to be restarted:

```
C:\>net stop w32time && net start w32time
```

The settings of the current time service as described in [10], can be listed with following command:

```
C:\>w32tm /dumpreg /subkey:config
```

By default the time client performs periodical checks every 45 minutes until time synchronization has been successful three consecutive times; then the period is set to once every 8 hours [9].

Meinberg[12] provides a more sophisticated NTP client and server, both hardware and software. Meinberg's NTP Time Server Monitor can be useful for monitoring time throughout applications. To get an idea of the contents of a NTP packet it is sufficient to capture packets with ethereal (Figure 2.8).

---

8 Note that you either have to know the local name server or enable DHCP. If DHCP is enabled can be checked with command `ipconfig -all`. Listed commands work with an enabled DHCP.

9 Note: When searching "ntp1" you are searching for server, which contains "ntp1" as a subdomain name (so any "ntp1" in the second-level domain, e.g. "kth.se", will be found). It does not have to be a NTP server, but it is very likely.

10 One or more NTP servers can be specified, each separated by a comma (without any spaces). As described in section 2.5, time should be synchronized based upon several servers.

11 This shows the registry entries of the Windows Time services with the stated subkey.

12 Meinberg Funkuhren; for more information see http://www.meinberg.de/.

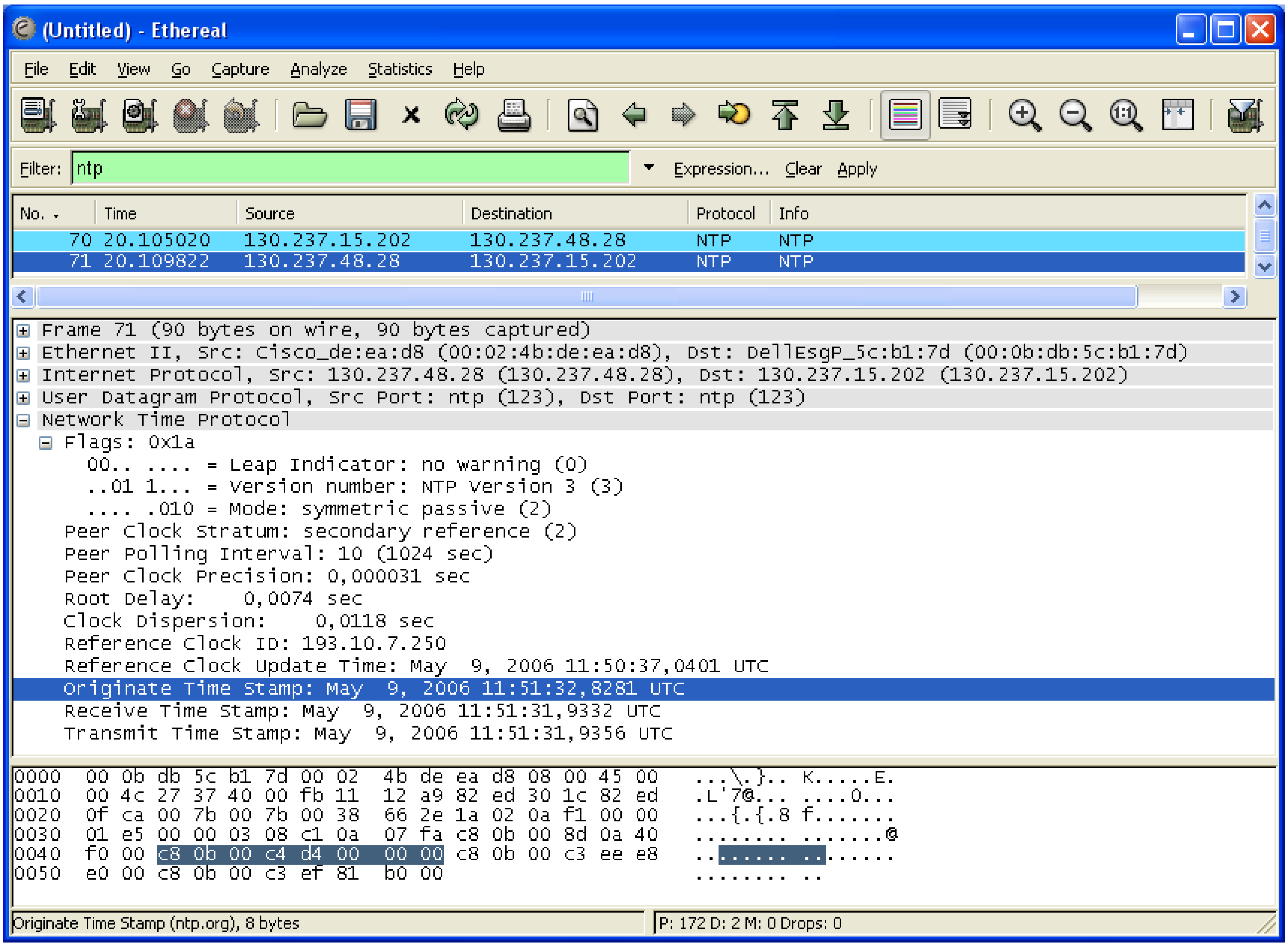

Figure 2.8: Snapshot of captured NTP packets in Ethereal, the highlighted field is the originate timestamp[13]

2.6 RTP Control Protocol – RTCP

In real-time applications it is essential for the senders to understand packet loss and delay. Therefore a primary function of the RTP control protocol (RTCP) is to provide feedback on the quality of the corresponding RTP session. To enable these two streams to be easily associated RTCP runs on another port and this port by convention is one greater than corresponding RTP port number.

RTCP defines the following packet types [17]:

- Sender Report – SR
- Receiver Report – RR
- Source Description Items – SDES
- End of participaction – BYE
- Application-specific functions – APP

Each of these will be described in the following subsections.

13 How to interpret this timestamp is covered in section 3.4.

2.6.1 Sender Report – SR

The sender report provides statistics of transmission and reception from participants that are active senders, which actively transmits RTP packets; in an interactive call both are active senders, but in case of a radio only the radio station is an active sender.

Table 2.3: RTCP Sender Report (SR) Packet [17]

	0			8	16	31
header	V=2	P	RC	PT=SR=200	length	
	SSRC of sender					
sender info	NTP timestamp, most significant word (MSW)					
	NTP timestamp, least significant word (LSW)					
	RTP timestamp					
	sender's packet count					
	sender's octet count					
report block 1	SSRC_1 (SSRC of first source)					
	fraction lost				cumulative number of packets lost	
	extended highest sequence number received					
	interarrival jitter					
	last SR (LSR)					
	delay since last SR (DLSR)					
report block 2	SSRC_2 (SSRC of second source)					
	...					
extensions	profile-specific extensions					

The SR packet consists of three sections:

Section 1: Header (64 bits / 2 words)

V (2 bits)	version, identifies the version of RTP, currently 2
P (1 bit)	padding bit
RC (5 bits)	reception report count
PT (8 bits)	packet type, for Sender Report it is 200
length (16 bits)	the length of the RTCP packet in 32-bit words minus one

SSRC (32 bits) synchronization source identifier for the originator of this SR packet.

Section 2: Sender Info (160 bits / 5 words)

The second section, the *sender info* summarizes the data transmission from the sender (SSRC), mentioned in the *header*. It provides the following information:

NTP timestamp (64 bits) local clock time as a NTP timestamp (see Section 2.5) with MSW representing seconds and LSW representing microseconds

RTP timestamp (32 bit) corresponds to the same time as the NTP timestamp (above), but in the same units and with the same random offset as the RTP timestamps in data packets

sender's packet count (32 bits) total number of RTP data packets transmitted by the sender

sender's octet count (32 bits) total number of payload octets (i.e., not including header or padding) transmitted in RTP data packets

Section 3: Report Block(s) (192 bits / 6 words for each block)

The third section contains zero or more reception *report blocks* providing the information listed below:

SSRC_n (32 bits) source identifier, where n stands for the n[th] reception report block

fraction lost (8 bit) fraction of RTP data packets lost since the previous SR

$$fraction\ lost = \frac{packets_{lost}}{packets_{expected}} \tag{2.1}$$

cumulative number of packets lost (24 bits) cumulative number of packets lost since reception has begun

$$cumulative\ packets\ lost = packets_{expected} - packets_{received} \tag{2.2}$$

extended highest sequence number received (32 bits) extended highest sequence number received; the first 16 bits contain the *sequence number cycle*[14] of the corresponding RTP sequence number, which are represented by the last 16 bit

interarrival jitter (32 bits) The inter arrival jitter (network delay variation) estimates the statistical variance of the delay between each RTP packet, expressed in RTP timestamp units.

First the difference *D* between RTP packets sent and arrived has to be calculated:

$$D(i-1, i) = D_{arrived\ packets} - D_{sent\ packets} = (R_{i-1} - R_i) - (S_{i-1} - S_i) \tag{2.3}$$

where:
R_i is the time (in RTP timestamp units) of the arrival of packet *i*
S_i is the RTP timestamp of packet *i*

After that the interarrival jitter *J* should be calculated for each data packet *i* arrived from source SSRC_n:

14 When the RTP sequence number get to its end, there will be another RTP sequence number generated; these cycles of new initiation of RTP sequence numbers will be stored in the variable of *sequence number cycle*. Effectively this extended the sequence number field to 32 bits (i.e., 16 + 16).

$$J(i) = \frac{J(i-1) + (|D(i-1,i)| - J(i-1))}{16} \tag{2.4}$$

where:
J(i) is the current inter arrival jitter value

The jitter **must** be calculated as shown in Formula 2.4 to allow profile-independent monitors [17].

LSR (32 bits) last SR timestamp; time (middle 32 bit of NTP timestamp) of the most recent RTCP sender report (SR)

DLSR (32 bits) delay since last SR expressed in units of 1/65536 seconds, between receiving the last SR packet from source SSRC_n and sending this reception report block; if no SR has been received yet, the DLSR-field is set to zero

2.6.2 Further RTCP Reports

Receiver Report – RR

Provides statistics of reception from participants that are not active senders. The format is the same as the SR packet except that the sender info is missing and the packet type field is set to 201 (i.e. Receiver Report).

Source Description Items – SDES

The SDES packet contains of a header and zero or more chunks containing SDES items. There are several possible source descriptions, starting with CNAME (Canonical End-Point Identifier), NAME, EMAIL, PHONE, and a few more. The CNAME item is very important, because it should provide a unique and a persistent transport-level identifier of the sender's source. It should be derived algorithmically with the format *user@host*, so that a third-party (e.g. the service provider) can monitor the flow of RTP packets[15].

BYE: Indicates end of participation

The BYE packet indicates that one or more sources are no longer active [17].

APP: Application-specific functions

The APP packet should be used for testing and application-specific functions [17].

2.6.3 RTCP Packet Format

As shown in Figure 2.9 a RTCP compound packet can contain several different RTCP packets and is enveloped in a lower layer protocol, such as UDP. The compound packet has to start with a report packet (SR or RR) and it has to contain a SDES packet with a valid CNAME. Furthermore there should only be one compound packet per report interval and an implementation of RFC-3550 should ignore incoming RTCP packet with unknown types [17].

---

15 Note that this assumes that the RTCP packets are not encrypted or tunneled.

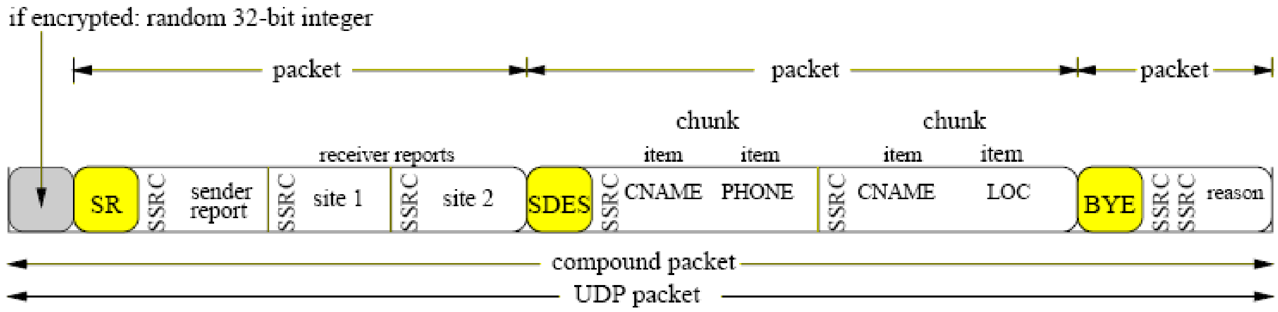

Figure 2.9: Example of a RTCP compound packet [17]

2.6.4 RTCP Transmission Interval

The RTCP traffic should be **small** relative to the primary function (sending RTP data packets) and should be **known** for each participant, so that it can be included in the bandwidth specification.

$$SendingInterval_{minimum}(\text{in sec}) = \frac{360}{\text{bandwidth in kilobits per sec}} \qquad (2.5)$$

The fixed minimum interval of sending RTCP packets should be 5 seconds, but it should be adjusted to the bandwidth of the RTCP sender as shown in Formula 2.5. So the minimum can be less than 5 seconds for a high bandwidth connection [17]. As it can be seen with Ethereal, the throughput of RTP packets is 10,700 bytes per second (50 packets with 214 bytes each), whereas a RTCP packet containing only a Sender Report has only 94 bytes (18,8 bytes per second with a 5-seconds interval) [21].

2.7 RTP Control Protocol Extended Reports – RTCP XR

The RTP Control Protocol Extended Reports (RTCP XR) enhances RTCP. RTCP XR defines seven block types, which can be divided into three categories:

A. Packet-by-packet block types
   1. Loss Run Length Encoding (RLE) Report Block
   2. Duplicate RLE Report Block
   3. Packet Receipt Times Report Block
B. Reference time information block types
   1. Receiver Reference Time Report Block
   2. Delay since last Receiver Report (DLRR) Report Block
C. Summary metric block types
   1. Statistics Summary Report Block
   2. VoIP Metrics Report Block

In this thesis we will only introduce categories B and C; for further information about category A see [7].

Table 2.4: XR Packet Format [7]

0 8 16 31

V=2	P	reserved	PT=XR=207	length
SSRC of sender				
report block(s)				

An XR packet consists of a RTP-version field, a padding flag, 5 bits reserved for future definition, packet type field (XR = 207), a length field, SSRC field, and report block(s).

Table 2.5: Format of an extended report block [7]

0 8 16 31

BT	type-specific	block length
type-specific block contents		

Each report block consist of a 8-bit block type field (BT), another 8-bit type-specific definition, a 16bit block length field, and the type-specific block contents, which are as follows [7]:

The *Receiver Reference Time Report Block* (BT=4) allows non-senders to send NTP timestamps, which indicates their wall clock time when the block was sent.

The *DLRR Report Block* (delay since the last *Receiver Report*, BT=5) extends RTCP, so that non-senders can calculate *round trip times* (RTT). It extends RCTP's DLSR mechanism and uses a similar format.

The *Statistics Summary Report Block* (BT=6) carries additional information about lost packets, jitter measurements, and time-to-live (TTL) values, which can be useful for network management.

The *VoIP Metrics Report Block* (BT=7) provides packet loss and discard metrics, delay metrics, analog metrics, and more.

3 The program "minisip"

3.1 Using minisip

As introduced in section 2.1, *minisip* is a softphone application for calling SIP-based phones or other computers with an SIP application, i.e. it is a SIP user agent (UA). Under Microsoft's Windows operating system minisip is currently available with a command line interface; just recently a user interface version has become available.

Before starting it is necessary to configure the SIP settings in the configuration file "*.minisip.conf*" (it should be in *[Project-Folder]\Project\minisip\debug*). In this configuration file you have to specify your SIP proxy (e.g. *sip:kth@iptel.org*[16]), your user name (e.g. *kth*) and password. There are many variables in *.minisip.conf*, such as CODECs, UDP, and TCP ports, and much more that can be specified to control the configuration of this client; for a detailed sample configuration file see Listing 8 in the Appendix.

After starting minisip it should display the following message:

```
Register to proxy iptel.org OK
IDLE$
```

Now you can use the `call`-command to call a SIP phone of your choice, for example:

```
IDLE$ call cisco1@130.237.15.222
```

16 You can register for an account at the SIP proxy http://www.iptel.org/.

3.2 Minisip's architecture

This section introduces the basic architecture and essential procedures of minisip.

3.2.1 Start-up

As it can be seen in Figure 3.1, minisip will first create **`SipSoftPhoneConfiguration`** and **`MinisipTextUI`** with its constructor (**`Minisip::Minisip`**). **`SipSoftPhoneConfiguration`** is used as a global container for objects (e.g. a new **`NTP`** object) and global variables.

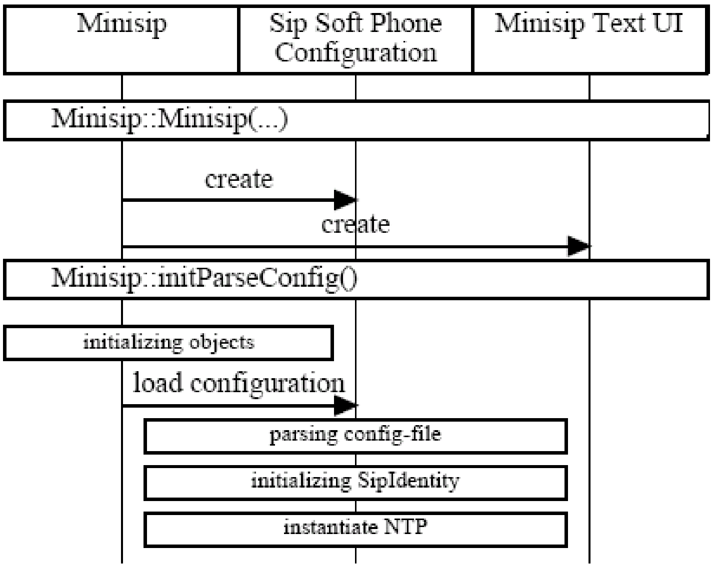

Figure 3.1: minisip start-up sequence

In **`Minisip::initParseConfig()`** all necessary objects for registering a SIP softphone will be initialized; these objects are **`IpProvider`**, **`MediaHandler`**, **`Sip`**, and **`MessageRouter`**. Next the configuration file *.minisip.config* will be parsed and saved in a new **`SipIdentity`** and an object of class **`NTP`** is instantiated.

3.2.2 Calling Procedure

From objects **`Sip`** or **`DefaultDialogHandler`** the **`Session`** will be started for each call by **`MediaHandler`**, which is the key class for handling media streams in minisip. As depicted in Figure 3.2 the **`MediaHandler`** will first create a **`Session`** object and then initialize all necessary objects for sending and receiving RTP and RTCP packets.

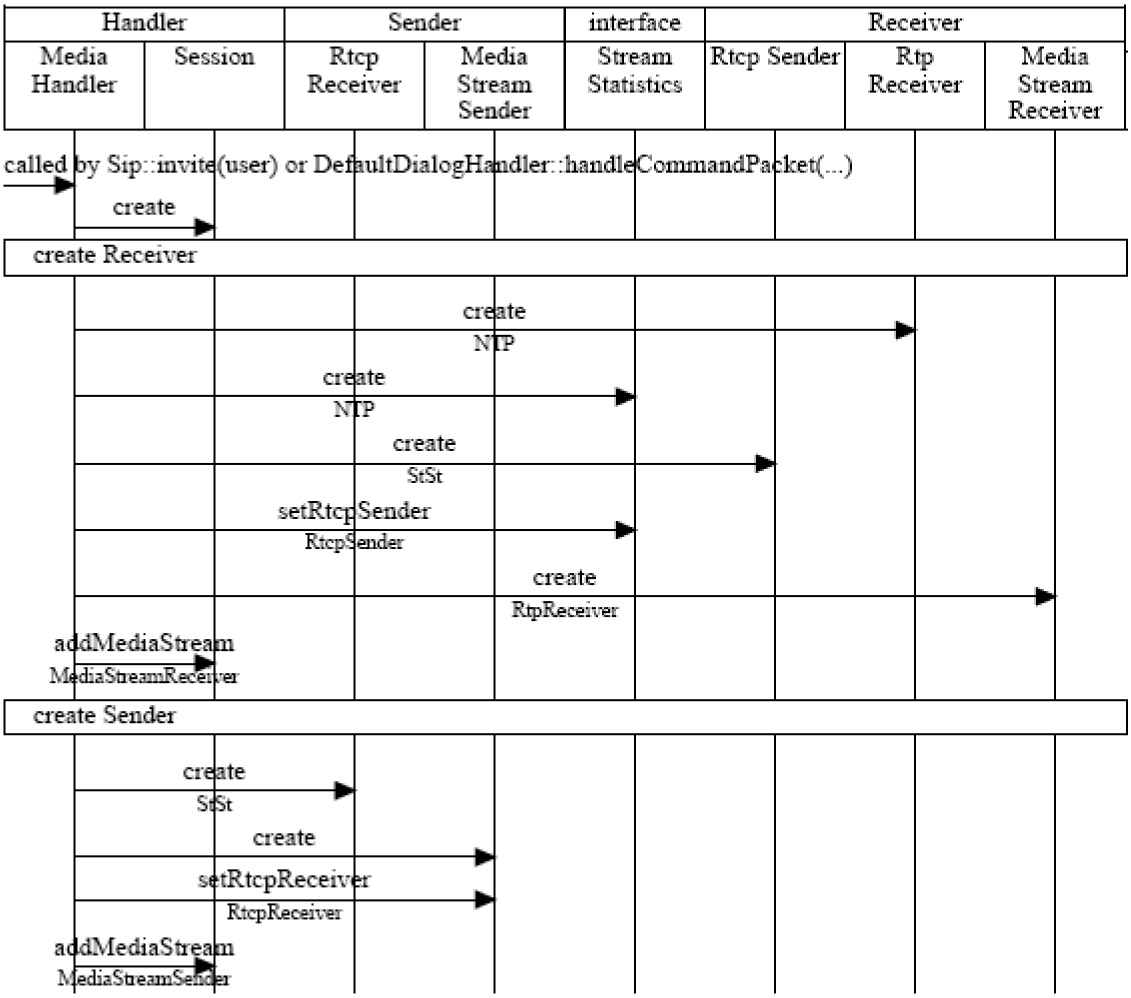

Figure 3.2: Call-setup procedure, where
StSt is the object of class StreamStatistics, and
NTP is the object of class NTP (which is stored in SipSoftPhoneConfiguration).

Figure 3.3 shows how minisip handles incoming and outgoing (S)RTP packets. A **`RtpReceiver`** is instantiated for each incoming source (SSRC), but it can have one or more media streams (**`MediaStreamReceiver`**). When a peer is sending both audio and video streams, minisip has to handle two different streams from the same source.

On the sender's side there is only the MediaStreamSender, because minisip does not care if 2 streams are related to each other.

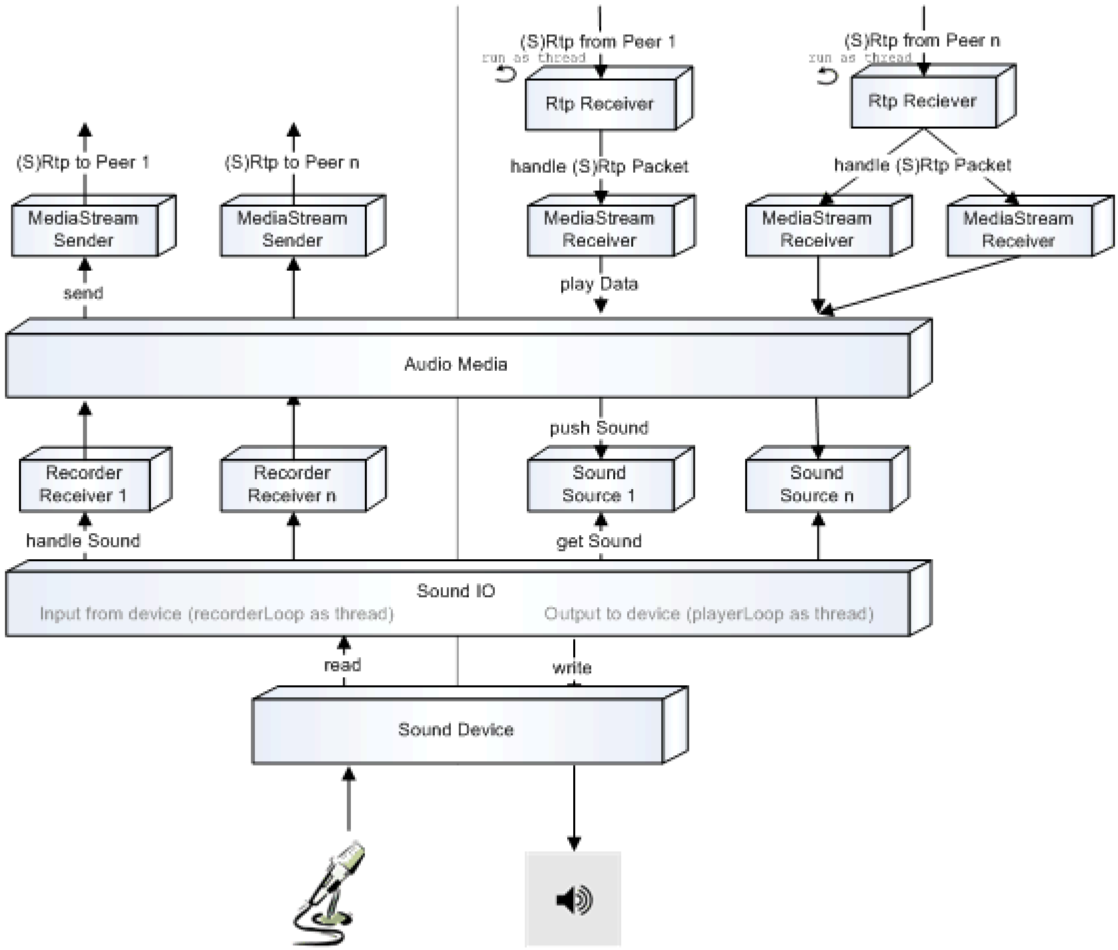

Figure 3.3: Audio Media System of minisip; figure based on [21] and [5]

Figure 3.4 shows the upper level of Figure 3.3 in more detail. Besides **`RtcpSender`** and **`RtcpReceiver`** another class (**`StreamStatistics`**) is needed to collect information of incoming and outgoing streams for RTCP reports. **`RtcpSRManager`** processes each received RTP packet and collects information necessary for the next RTCP packet. When the RTCP interval has been reached **`RtcpSRManager`** gets information concerning sent RTP packets through **`StreamStatistics`** and saves the necessary statistics for **`RtcpReceiver`**. When the RTCP packet (for a full class diagram of RTCP packet see Figure A.2) has been built it will be sent by **`RtcpSender`**.

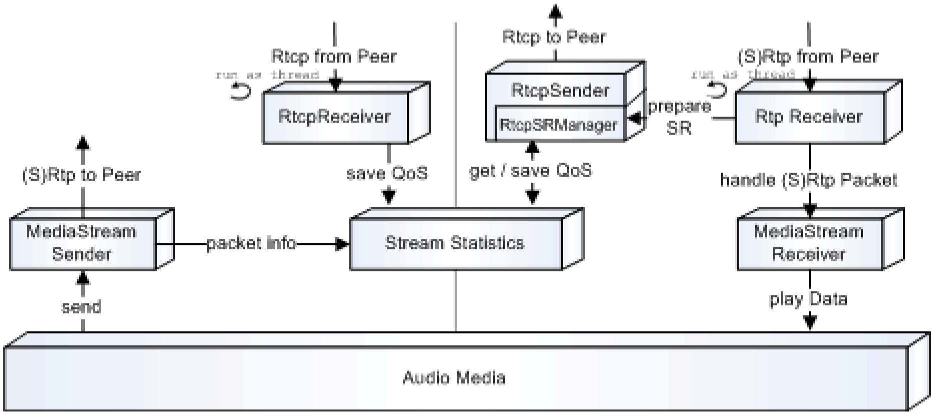

Figure 3.4: Audio Media System including RTCP

3.2.3 RTCP Packet Structure

Minisip uses different naming for the RTCP compound packet and RTCP packet than those used in RFC-3550 [17], as introduced in section 2.6.3. Table 3.1 shows the corresponding names.

Table 3.1: Names as used in minisip vs. RFC-3550

Naming in minisip	***Naming in RFC-3550***
RtcpPacket	RTCP compound packet (see 2.6.3)
RtcpReport	RTCP packet (see 2.6.3)
RtcpReportReceptionBlock	Report block (see 2.6.1)

Figure 3.5 shows the RTCP packet structure in minisip[17]. All classes in grey are implemented and used by minisip, whereas classes with white background are available, but not yet fully implemented.

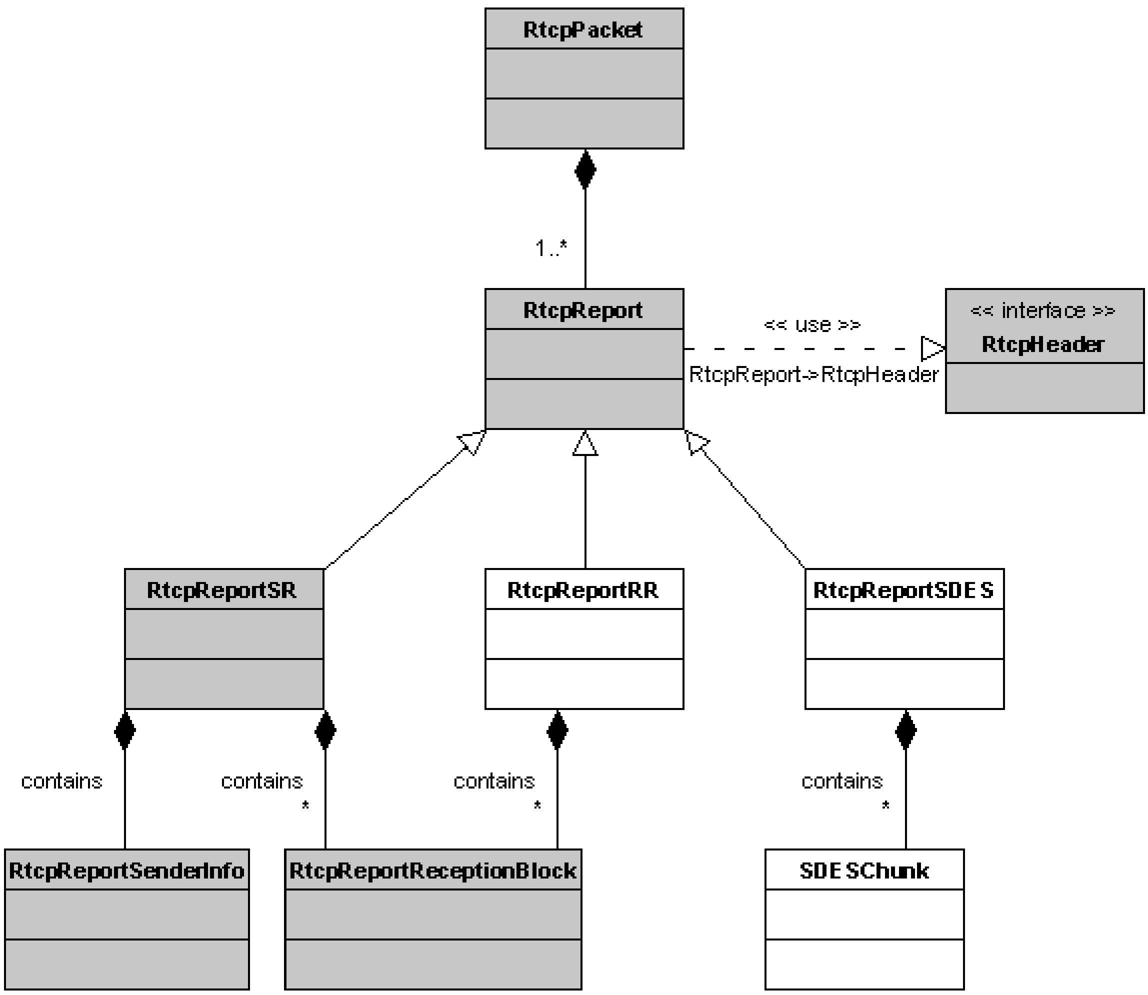

Figure 3.5: RTCP Packet Structure in minisip

17 A RTCP Packet Structure including SDES in more detail can be seen in Figure A.2 in the Appendix.

RtcpPacket	An object class for the compound RTCP packet as described in RFC-3550 [17].
RtcpReport	Inherits RtcpHeader and is inherited by various kind of report types, such as RtcpReportSR, RtcpReportRR.
RtcpHeader	Represents the header section of a RTCP report and is inherited by RtcpReport.
RtcpReportSR	Represents the Sender Report of RTCP. It inherits the RtcpReport and thus also RtcpHeader.
RtcpReportRR	Same structure as RtcpReportSR, but without the sender info.
RtcpReportSDES	Represents the source description (SDES) of RTCP. It is inherited by SDESChunk, which can be of a specific kind of SDES, such as CNAME, EMAIL, etc. For a more detailed diagram in the sense of SDES, see Figure A.2 in the Appendix.
RtcpReportSenderInfo	Represents the sender info section of a RTCP Sender Report.
RtcpReport ReceptionBlock	Represents the report block of a RTCP report.

3.3 RTP sequence order and packet loss

While calculating packet loss might at first seem to be a very simple computation there are several things, which have to be considered when calculating packet loss:

1. sequence of RTP packets and their potential missorder (i.e., when they are received in the wrong sequence)
2. inter packet jitter
3. new RTP sequence number cycle
4. real packet loss

The above points can also occur together; for example after a (long) inter packet jitter a packet arrives in the wrong order (i.e., out of sequence), as depicted in Figure 3.6.

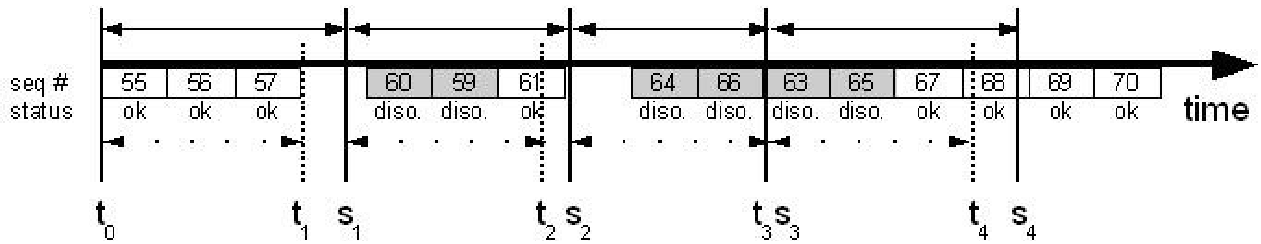

Figure 3.6: Exemplary time line for arrived RTP packets, where
t_x is the exact timeout time
s_x is the time sending a RTCP packet

Table 3.2: Exemplary time table based on Figure 3.6

RTCP packet sent	***received***	***expected***	***cum. lost***	***fraction lost***
s_1	3	3	0	0
s_2	6	7	1	64
s_3	8	12	4	154
s_4	11	13	2	0

First three packets are received in order at the receiver without any inter arrival jitter. The next packet (#60), however, arrives after the last packet in order (#57). Packet 60 is out of order, because it precedes packets 58 and 59, where the latter packet finally arrives after packet 60; therefore we only actually lost a single packet. The fraction lost is calculated as shown in equation 3.1:

$$fraction\,lost=\frac{cummulative\,loss_{s2}-cummulative\,loss_{s1}}{expected\ packets_{s2}-expected\ packets_{s1}}*256=\frac{1}{4}*256=64 \quad (3.1)$$

Since fraction lost is represented as an 8 bit integer representing the fractional loss in units of 1/256, the fraction in equation 3.1 has to be multiplied by 256. At time s_3 minisip would expect five more packets (#62 - #66), but only two packets have been received, so there is a packet loss of three packets since s_2; since the start of the session five packets have been lost (cumulative lost). At time s_4 minisip get more packets than expected, thus the fraction lost would be negative, but a negative fractional loss is not suitable and has to be replaced by a zero fraction loss. Note also that the cumulative loss is decreasing and therefore correcting itself, because packets might arrive late (such as packets 63 and 65 at t_3), but the cumulative loss will be corrected next time (in this example at t_4).

3.4 Using NTP in minisip

As described in Section 2.5, all NTP timestamps are represented by an unsigned 64-bit fixed point number. Table 3.3 shows how time is encoded in NTP format.

Table 3.3: Structure of NTP timestamps

0	15	31	47	63
integer-part (MSW)			fraction-part (LSW)	

The time shown in Figure 2.8 (May 9, 2006 11:51:32,8281 UTC) in NTP timestamp format is encoded as (shown in binary, hexadecimal, and decimal):

`1100 1000 0000 1011 0000 0000 1100 0100`	`1101 0100 0000 0000 0000 0000 0000 0000`
`C 8 0 B 0 0 C 4`	`D 4 0 0 0 0 0 0`
`3,356,164,292 (seconds)`	`3,556,769,792` (2^{-32} `seconds)`

The integer-part can just be handled as an ordinary 32-bit integer and represents seconds from 1900-01-01, 0:00:00,0000 UTC. The Least Significant Word (LSW) represents the fraction of a second, measured in 2^{-32} seconds; equation 3.2 shows how to convert from LSW to microseconds (µs):

$$s = LSW * \frac{1}{2^{32}} = 3{,}556{,}769{,}792 * \frac{1}{4{,}294{,}967{,}296} = 0.828125\, seconds \tag{3.2}$$

For RTCP the middle 32-bits of the NTP timestamp is needed for last sender report (LSR) and delay since last sender report (DLSR) (see section 2.6.1); the LSR value of the time in Table 3.3 would be 12,899,328 (hex: `00C4 D400`).

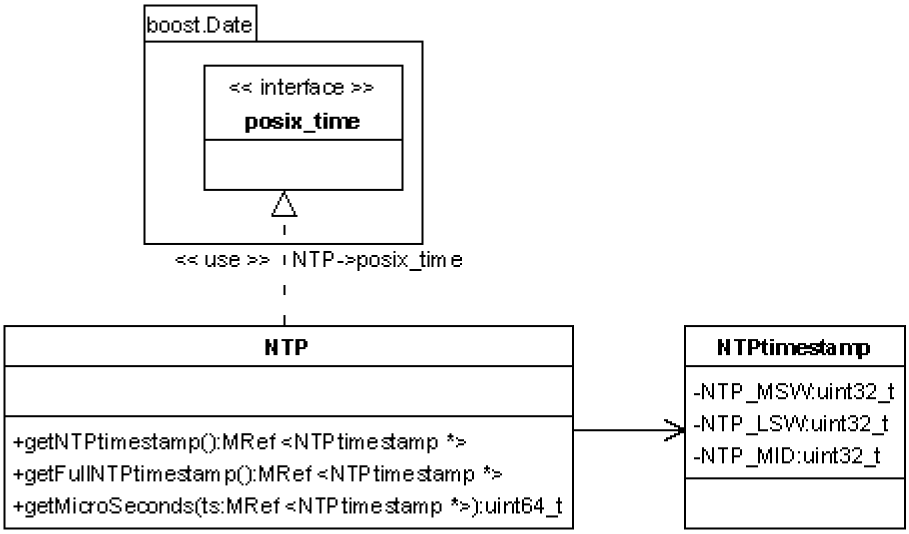

Figure 3.7: NTP class structure

As depicted in Figure 3.7, the NTP timestamp is stored in an object class, named **`NTPtimestamp`**, which is instantiated by the class **`NTP`**. Methods **`getFullNTPtimestamp()`** and **`getNTPtimestamp()`** both return a Memory Reference (**`MRef`**), which is minisip's Smart Pointer implementation [12]. Using template class **`MRef`** it is possible to check types of pointers and therefore this is a safe way to access these references.

Class **`NTP`** gets a current time in microseconds from the package **`boost.Date`** [2], then converts this into NTP format and saves it in a new object of type **`NTPtimestamp`**. Formerly **`ibmts`** – an IBM[18] timestamp – was used, but this has been depreciated, since it is “only” a timestamp and does not implicitly get an accurate UTC time. Timestamps are normally used for comparision of two or more points in time. Since we have different clocks in VoIP it is necessary to use a global time to accurately compare times. Listing 3 in Appendix C.1 shows the C++ code for doing all computation described in this section.

The smallest resolution of time provided by the built-in Real-Time Clock (RTC) in the computer used for this thesis work, ccsser2, which is an Intel® 82801DB LPC Interface Controller 24C0, is around 122 µs[19]. Therefore minisip can only provide an accuracy of about one eight millisecond on an Intel® Xeon™ CPU running at 2.80GHz [8].

18 IBM™ (International Business Machines) is a registered name; see http://www.ibm.com/ for more information

19 Machine ccsser2 is a Dell Precision 450 workstation.

4 Testing

4.1 Test setup

There are two machines (A and B) in this test, each is running minisip. The TCP/UDP traffic goes through a NIST Net server[20], which forwards every packet to the other peer (see Figure 4.1) [13].

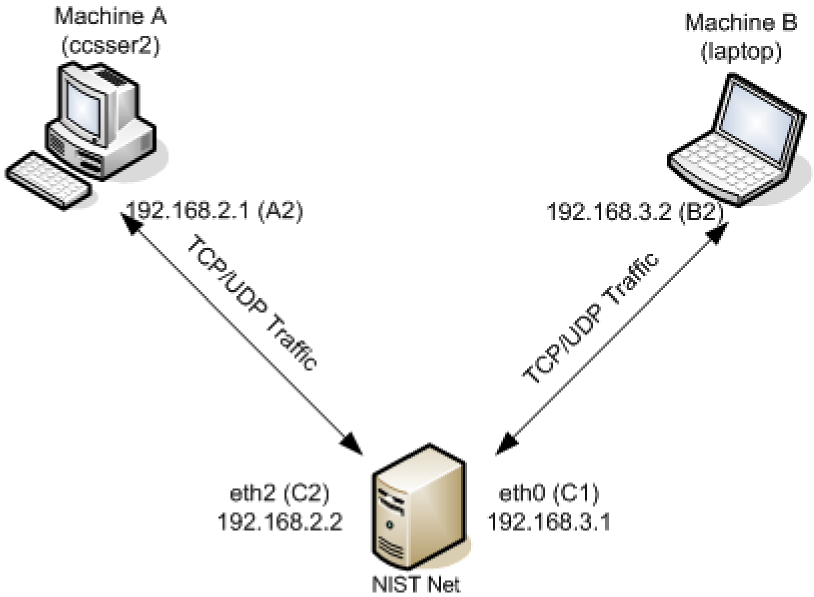

Figure 4.1: Test setup with a NIST Net server

The NIST Net server has been configured to drop a certain amount of packets[21]:

```
cnistnet -a b2 a2 --drop 7 --up
cnistnet -a a2 b2 --drop 1 --up
```

Thus approximately seven per cent of all packets from machine B (interface B2) to A (interface A2) will be dropped by NIST Net and one per cent on the reverse path. The value of 1% has been chosen to check if minisip will detect even small amounts of packet loss, and with an quite high 7% of loss we can check if the packet loss calculations are correct.

20 For a step-by-step instruction for test setup, see Appendix C.1.

21 To check if nistnet has been configured correctly type command `cnistnet -R`.

4.2 Test results

This test was conducted to check if all fields of the RTCP packets have been computed correctly. The most significant values are listed in Table 4.1, as extracted by Ethereal.

RTCP #	source	fraction lost		cum. lost	DLSR		remark
		in 1/256	in %		in 1/65536 s	in ms	
1	B2	2	1.04%	2	0	0	1st RTCP packet
1	A2	18	7.29%	14	3 072	47	
2	B2	1	0.40%	3	325 526	4 967	
2	A2	20	7.41%	33	4 096	63	
3	B2	0	0.00%	3	324 214	4 947	not transmitted
3	A2	19	7.60%	52	331 776	5 063	
4	B2	0	0,00%	3	326 183	4 977	
4	A2	19	7.60%	71	1 024	16	

Table 4.1: Test results

This test validated the field values in each RTCP packet. This validation was done by comparing statistics provided by RTCP packets and by Ethereal RTP stream analysis; in addition to the displayed fields, LSR and sender's packet count have been checked. RTCP packet # 3 from B2 was dropped by NIST Net; to recognize loss of RTCP packets, the other peer has to check the DLSR field. Therefore in RTCP packet #3 of A2 the DLSR is much higher than previously, because peer A2 did not receive RTCP #3 from B2.

Since the values transmitted by RTCP has been identical with values captured by Ethereal, the underlying RTCP code is working properly. A reason for a zero packet loss in packet #3 and #4 of B2 might be that repeated packets has been dropped by NIST Net.

Figure 4.2 shows RTCP packet # 4 of A2 as captured by Ethereal.

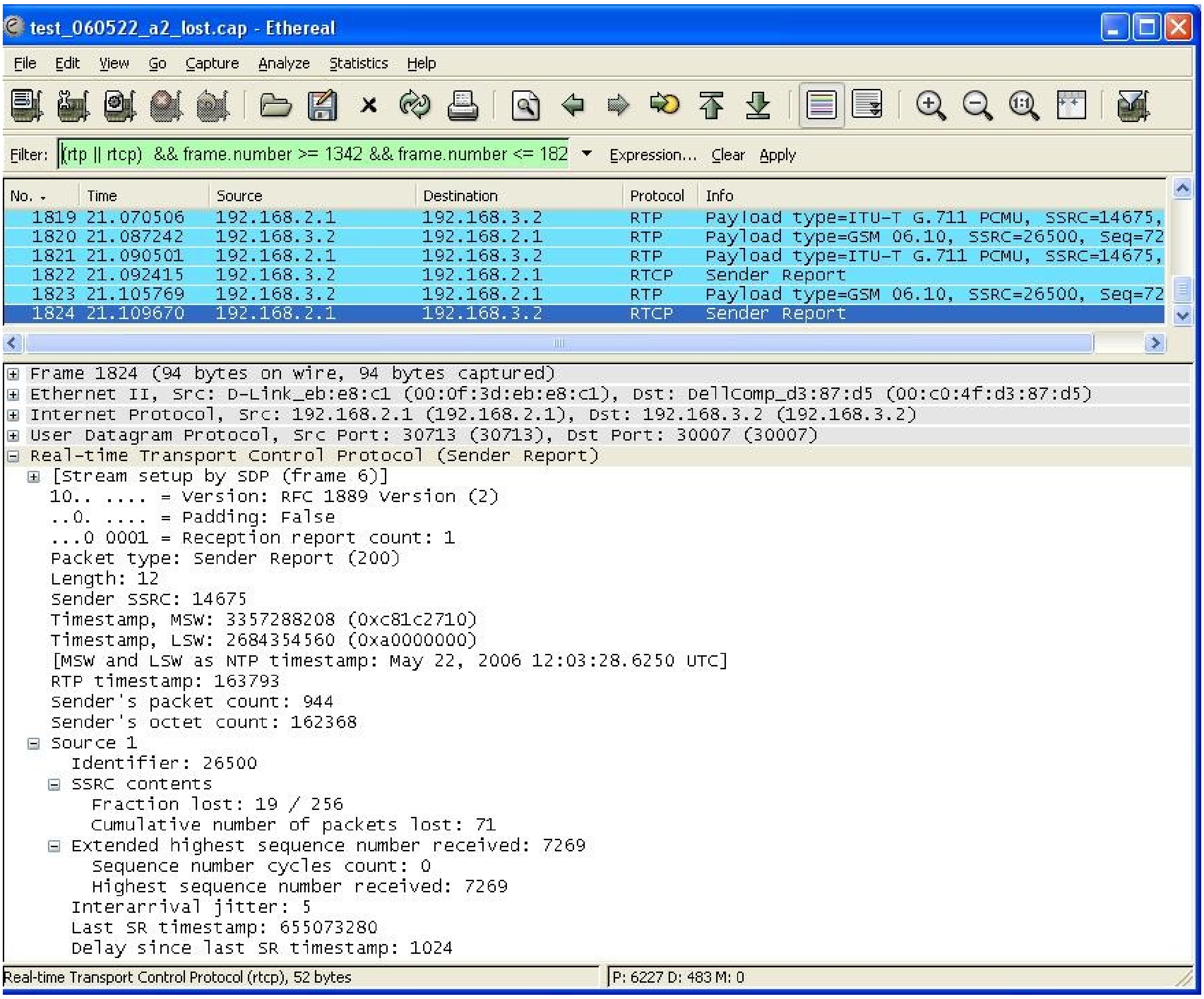

Figure 4.2: RTCP packet in ethereal

4.3 Further tests

With minisip 0.7.1 for Windows XP it is possible to make conference calls. The code which this thesis is based on, however, is not yet able to make conference calls. Minisip kills the running process with a “Debug error”, but the location and reason for this error could not be determined. Since the minisip “trunk” is the current version most developers are working on and conference calls are already implemented, it is reasonable to move the RTCP code into trunk. While previous Master theses have been of a more experimental nature and the code was ported from using the GNU compiler to using Microsoft's Visual Studio, the version of the code used in this thesis and that of Xiaokun Yi has been saved in its own branch. Since this code branch was created, a lot of development work has taken place on the trunk, hence implementing RTCP in trunk would enable conference calls more easily than trying to understand the problems in the code underlying this thesis. According to Erik Eliasson, a lead programmer of minisip, several changes are needed before the RTCP code can be implemented in the trunk. Therefore the integration of RTCP code in the current trunk version should be done by one of the minisip developers.

A video call would have been a valuable test especially for QoS, because minisip has to distinguish between audio and video data. As it can be seen in Figure 3.3, minisip creates two **`MediaStreamReceiver`** objects when receiving a video call: one stream for audio and one for video. However, there is only one instance of **`RtpReceiver`** and therefore only one **`RtcpSender`**, which means minisip calculates receiving statistics for each sender source (SSRC) and **not** for each received stream! Since audio and video streams can transmitted through different network routes, it may happen that the audio stream loses a lot of packets, while video has a perfect connection. This situation cannot yet be distinguished by minisip[22].

22 Note that it could not found out, how Skype or eyeBeam are handling this issue.

5 Conclusion and Future Work

5.1 Conclusion

The objective of this thesis was to examine how to exploit a sense of global time in Voice over IP. It is clear that this global time can be used for Quality of Service (QoS) and synchronization of streams. This thesis has focused on the first of these uses in the context of RTCP.

NTP was selected as a global timebase, since it is a standardized protocol for over two decades and provides all necessary mechanisms for getting knowledge of accurate global time. Since spyware and hackers try to use standard ports, such as NTP's 123, to get through firewalls, even an internet standard, such as NTP, might cause a security leak. Therefore most intranets have NTP proxies, which synchronize with other NTP servers (see section 2.5). As a result there has to be either a proxy NTP server inside the intranet or a GPS receiver can be used to locally get the correct global time for use by an NTP server inside the intranet.

A full implementation of RTCP, which provides a valuable feedback mechanism for RTP streams, requires NTP for analyzing delays and jitter. A big problem of RTCP, however, is frequently an incomplete or limited implementation[23] in VoIP software. This thesis provides a full implementation of RTCP's Sender Report (SR), which is sufficient for an ordinary person-to-person call. A challenge when adding RTCP to minisip has been connecting both, sending and receiving, streams to provide the required statistical information (see section 3.2). This leads to the introduction of the Stream Statistic class - which can interact with both the senders and receivers of media streams.

5.2 Future Work

This thesis provides basic structures and methods for more sophisticated features, which are described below.

Integrate RTCP and NTP in minisip trunk

As noted in section 4.3, conference calls do not work with the underlying code base used for this thesis. Hence the next very important step involves integrating RTCP and NTP code into the current version (maintained by subversion (SVN)), in which conference calls are working. Though it sounds easy, it is not, because many people have changed quite a lot of code simultaneously. As a result this integration should be done by one of the main minisip developers. Afterwards it should be possible to check if RTCP is working properly in conference calls, video calls, and conference video calls.

Implement RTCP RR, SDES, and XR

The basic procedure of sending and receiving RTCP packets has been implemented, but it should be extended by enabling Receiver Reports (RR) for nodes that are only receivers and never senders, source description (SDES), and extended Reports (XR). RR itself should not be a big problem, because it has the same structure as SR, but without any sender info. The basic structure of SDES is depicted in Figure A.2, but has not been implemented. Fortunately SDES and XR are not essential

23 For example, in X-Lite's free version RTCP is not fully implemented. It is only implemented in the commercial version “eyeBeam” [3]. Skype [19] could have implemented RTCP fully, but this can only be analyzed by a Skype peer, as all of the Skype traffic is encrypted.

for QoS and only provides additional information about peers, but for a more sophisticated and user friendly interface both should be implemented.

Evaluate RTCP

Currently minisip considers only packet loss statistics from RTCP packets for CODEC switching (see [21]). Additionlly, RTCP could be used to adjust the playout buffer as described in section 2.4. This should be evaluated in a future thesis. Additionally, the user interface could alert the user when a certain amount of jitter or packet loss has been observed.

Synchronizing time through NTP automatically

Currently the NTP module only uses the value of the local clock, but does not check if this host's clock is synchronized with some accurate time source. To be able to synchronize time, minisip should discover potential NTP servers and then – if minisip has administrative rights – set the synchronization source for NTP, as was done manually in section 2.5.3. When the clocks of all participants provide accuracate time, then interstream synchronization could be realized, as described in section 2.4. This work should probably occur in combination with the above work concerning the playout buffer as the multiple sources could be synchronized via the time offsets used in [14].

References

[1] Banerjee, K.: "Introduction to RTP A Made Easy Tutorial", 2005
http://geocities.com/intro_to_multimedia/RTP/

[2] Boost community: "Boost C++ Libraries", May 2006 - http://www.boost.org/

[3] CounterPath: "CounterPath product page", May 2006 - http://www.xten.com/

[4] Ebert, Jean-Pierre: "Energy-efficient Communication in Ad Hoc Wireless Local Area Networks", Apr 2004 - http://edocs.tu-berlin.de/diss/2004/ebert_jeanpier

[5] Eliasson, Erik: "Minisip design overview", May 2006 - http://www.minisip.org/

[6] Elliot, Colm: "Stream Synchronization for Voice over IP Conference Bridges", 2004 - ftp://ftp.it.kth.se/Reports/DEGREE-PROJECT-REPORTS

[7] Friedman, et al.: "RTP Control Protocol Extended Reports (RTCP XR)", Jul 2003 - http://www.faqs.org/rfcs/rfc3611.html

[8] Intel: "Intel(R) 82801 DB I/O Controller Hub 4 (ICH4) Datasheet", May 2002

[9] Microsoft: "Basic Operation of the Windows Time Service", Nov 2003 - http://support.microsoft.com/?kbid=224799

[10] Microsoft: "Windows Time Service Tools and Settings", Mar 2003 - http://technet2.microsoft.com/WindowsServer/en/Lib

[11] Mills, David L.: "Network Time Protocol (Version 3)", Mar 1992
http://www.faqs.org/rfcs/rfc1305.html

[12] Minisip community: "minisip website", May 2006 - http://www.minisip.org/

[13] Nation Institute of Standards and Technology, U.S.: “NIST Net home page”, Jul 2005
http://snad.ncsl.nist.gov/nistnet/

[14] Pardo, Ignacio Sánchez: "Spatial Audio for the Mobile User", 2005 - ftp://ftp.it.kth.se/Reports/DEGREE-PROJECT-REPORTS

[15] Rosenberg, et. al.: "SIP: Session Initiation Protocol", Jun 2002 - http://www.faqs.org/rfcs/rfc3261.html

[16] Santasusana, Cardona: "MiniSIP Overview", Jun 2006 - http://www.minisip.org/doc/

[17] Schulzrinne, et al.: "RTP: A Transport Protocol for Real-Time Applications", Jul 2003 - http://www.faqs.org/rfcs/rfc3550.html

[18] Shih, et. al.: "Wake on Wireless: An Event Driven Energy Saving Strategy [...]", Sep 2002 - http://oscar.lcs.mit.edu/~eugene/research/papers/s

[19] Skype Limited: "Skype product page", May 2006 - http://www.skype.com

[20] Windl, et. al.: “The NTP FAQ and HOWTO”, Oct 2005
http://www.ntp.org/ntpfaq/NTP-a-faq.htm

[21] Yi, Xiaokun: "Adaptive Wireless Multimedia Services", May 2006 - ftp://ftp.it.kth.se/Reports/DEGREE-PROJECT-REPORTS

Appendix

Appendix A. Diagrams

This appendix contains additional and full page diagrams.

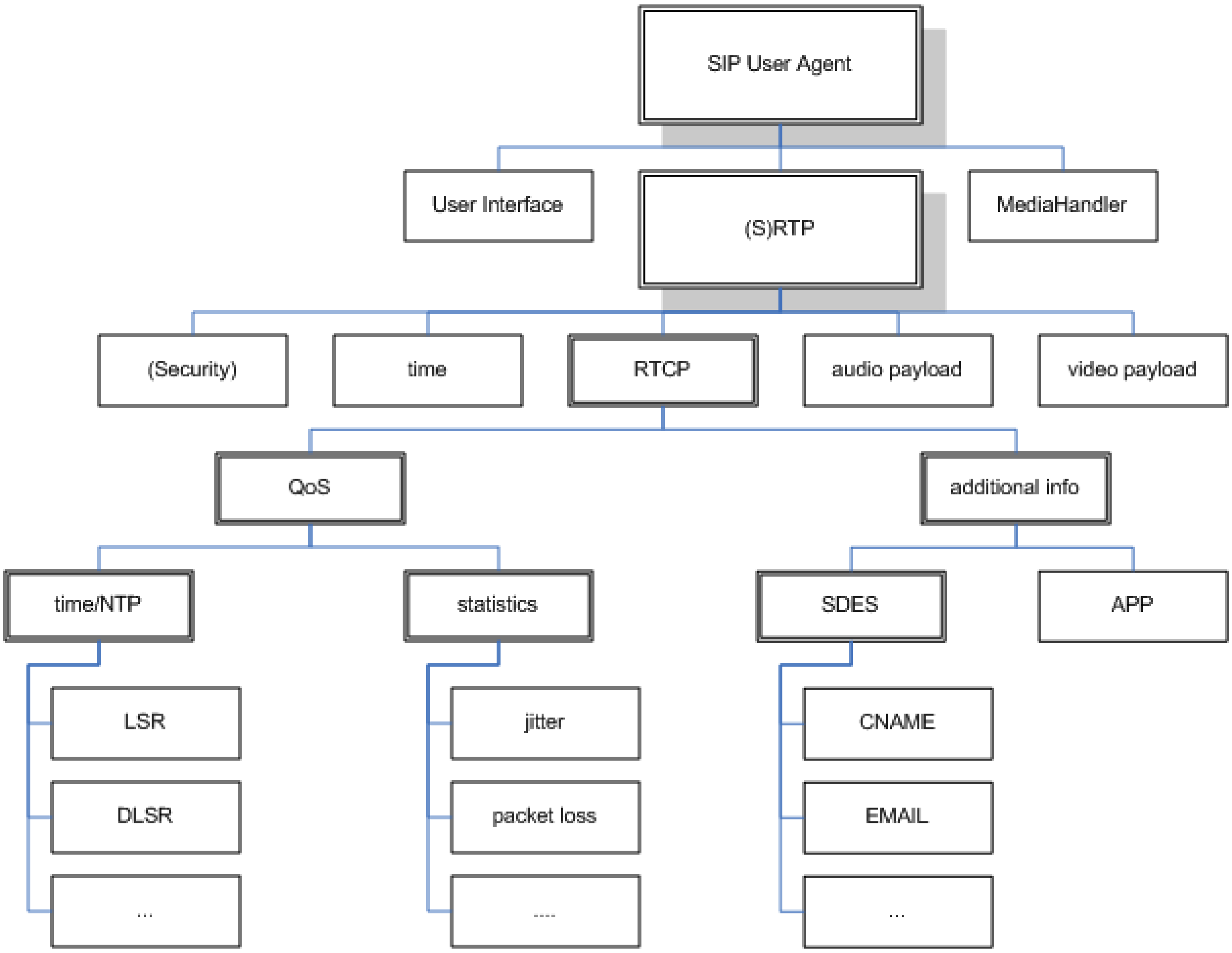

Figure A.1: Structure of an SIP User Agent, such as minisip, and its subsystems

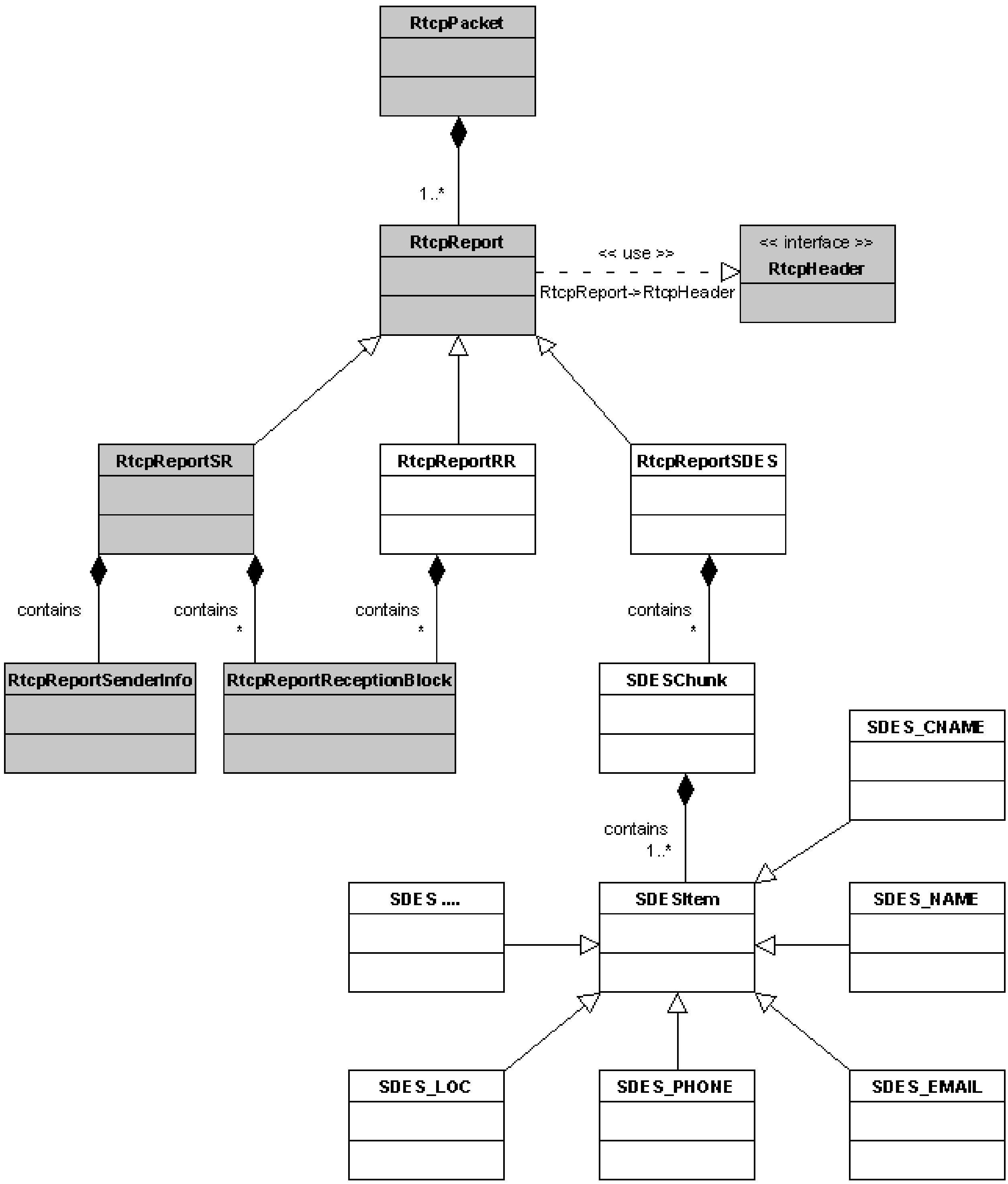

Figure A.2: RTCP Packet Structure in minisip including SDES structure in more detail

Appendix B. Minisip code excerpts

This appendix shows essential code excerpts for implementing the methods described in this thesis.

B.1 Excerpts of NTP and NTPtimestamp

```
/* to ensure that header file is not read twice ! */
#ifndef NTPTIMESTAMP_H
#define NTPTIMESTAMP_H

#include<config.h>
#include <libmutil/MemObject.h>

/** Object class representing NTP timestamps.
NTP_MSW - Most Significant Word: first 32 bits of NTP timestamp
NTP_LSW - Least Significant Word: second 32 bits of NTP timestamp
NTP_MID - mid 32 bits: last 16 bits of MSW and first 16 bits of LSW */
class NTPtimestamp : public MObject {
    public:
        /* public Constructors, Getters and Setters */
    private:
        uint32_t NTP_MSW;
        uint32_t NTP_LSW;
        uint32_t NTP_MID;
};

#endif
```

Listing 1: Header file of class NTPtimestamp

```
/* #ifndef and #includes */

class NTP : public MObject {
    public:
        /* public methods of class NTP are defined here */

    private:
        // 2^32 is equivalent to 4,294,967,296
        static const uint64_t NTP_MULTIPLIER = 4294967296;
        // from 1900 to 1970 it is 2,208,988,800 seconds
        static const uint32_t SECONDS_1900_1970 = 2208988800;
};

#endif
```

Listing 2: Header file of class NTP

```
/** Full NTP timestamp in MSW and LSW, including MID (middle 32 bits).
*/
```

```
MRef <NTPtimestamp *> NTP::getFullNTPtimestamp() {
    static ptime ptime_1970 = ptime(date(1970,1,1));
    static ptime now;
    static time_duration diff;

    static uint16_t MSW16;
    static uint16_t LSW16;
    static uint32_t MID;

    // get the current time from the clock in microsecond resolution
    now = microsec_clock::universal_time();

    diff = now - ptime_1970;

    MRef <NTPtimestamp *> retNTP = new NTPtimestamp (
        (uint32_t) (diff.total_seconds() + SECONDS_1900_1970),
        (uint32_t) (((diff.total_microseconds() -
     (diff.total_seconds() * 1000000)) * NTP_MULTIPLIER) / 1000000));

    // calculating middle 32-bits of NTP timestamp
    MSW16 = (uint16_t) retNTP->getNTP_MSW();
    LSW16 = (uint16_t) (retNTP->getNTP_LSW() >> 16);
    MID = 0;
    MID = MID | MSW16;
    MID = (MID << 16) | LSW16;
    retNTP->setNTP_MID(MID);

    return retNTP;
}
```

Listing 3: Method getFullNTPtimestamp() of class NTP

```
uint64_t NTP::getMicroSeconds(MRef <NTPtimestamp *> ts) {
    return (((uint64_t) ts->getNTP_MSW()) * 1000000) +
        (((uint64_t) (ts->getNTP_LSW())) * 1000000) /
            NTP_MULTIPLIER;
}
```

Listing 4: Method getMicroSeconds(...) of class NTP

B.2 Excerpts of RTCP code

```
void RtpReceiver::run() {
    MRef<SRtpPacket *> packet;

#ifdef RTCP_ENABLED
    IPAddress *remoteAddress=NULL;
    int32_t remotePort=0;
#ifdef NTP_ENABLED
    static MRef <NTPtimestamp *> currentTs = ntp->getNTPtimestamp();
#endif
#endif
```

```
    // begin while loop for receiving RTP packets
    while( !kill ) {
#ifdef RTCP_ENABLED
        static RtcpPacket *rtcpPacket;

/* the statement next line is NOT working at this point, because
rtcpSender has not been created yet in MediaHandler !!!
Therefore first timeout is just 1 second. */

// static uint32_t rtcpInterval = rtcpSender->getRtcpInterval();

        static uint64_t time = ntp->getMicroSeconds(currentTs);
        static uint64_t timeout = time + 1000000;

        // initialize more variables [...]

        currentTs = ntp->getNTPtimestamp();
        time = ntp->getMicroSeconds(currentTs);

        // call not yet answered or no RTP packet received
        if (time >= timeout && firstTime) {
            timeout = time + rtcpSender->getRtcpInterval();
        } else if (time >= timeout) {
            rtcpSender->sendRtcpPacket();
            currentTs = ntp->getNTPtimestamp();
            time = ntp->getMicroSeconds(currentTs);
            timeout = time + rtcpSender->getRtcpInterval();
        }
#endif // RTCP_ENABLED

        // code for receiving packets [...]
        // TO DO:
        // authenticate RTP packets; currently in MediaHandler

#ifdef RTCP_ENABLED
        // ***************************************
        // RTP packet arrived
        // calculate jitter, check seq-#, etc
        rtcpSender->getRtcpSRManager().prepareSR(packet, time);

        // first packet --> re-start timer [...]
#endif // RTCP_ENABLED

        // handle RTP packet [...]
    } // end while loop
} // end RtpReceiver::run
```

Listing 5: Method run() in class RtcpReceiver

```
void RtcpSRManager::jitterCalc(uint32_t RtpTimestamp, uint64_t time)
{    /** calculates jitter (max. 2147 seconds = 32 bit) */
     static uint32_t oldRtpTimestamp = RtpTimestamp;
     static uint64_t oldTime = time;
     static double D; // Deviation

// calculate deviation; 1 sample = 125 microsec.
     D = ((time - oldTime) / 125) - (RtpTimestamp - oldRtpTimestamp);

     this->jitter = ( this->jitter + (abs(D) - this->jitter) ) / 16;

     oldTime = time;
     oldRtpTimestamp = RtpTimestamp;
}
```

Listing 6: Method jitterCalc(..) in class RtcpSRManager

```
RtcpReportSR *RtcpSRManager::createSenderReport() {
     /** creates Sender Report (SR) */

     // Declarations [...]

     // Calculating lost and fraction lost
     this->expected = this->max_seq - this->base_seq + 1;
     this->cumulativeLost = (this->expected + this->expected_prior) -
(this->received + this->received_prior);
     lost = (this->expected + this->expected_prior - lastExpected) -
(this->received + this->received_prior - lastReceived);

     if ((this->expected - lastExpected) == 0 || lost <= 0) {
          fraction = 0;
     } else {
          fraction = (float) lost / (float) (this->expected + this-
>expected_prior - lastExpected);
     }
     this->fractionLost = (uint8_t) (fraction * 256);
     lastExpected = this->expected + this->expected_prior;
     lastReceived = this->received + this->received_prior;

     // Computing NTP timestamps
#ifdef NTP_ENABLED
     LSR = ts->getNTP_MID();
     ts = ntp->getFullNTPtimestamp();
     if (sStt->getLSR() != 0) {
          DLSR = ts->getMID_Diff(sStt->getLSR());
     }

     uint64_t rtp_diff = (ntp->calcDiff(ts, sStt->getNTPtimestamp()) /
125);

     RtcpReportSR *sr=new RtcpReportSR(*(new RtcpReportSenderInfo(
          sStt->getLastRtpTs() + rtp_diff,
          sStt->getSPacketCount(),
```

```
        sStt->getSOctetCount(),
        ts->getNTP_MSW(),
        ts->getNTP_LSW())
        ));
    RtcpReportReceptionBlock *rb=new RtcpReportReceptionBlock();
#else
    RtcpReportSR *sr=new RtcpReportSR(*(new RtcpReportSenderInfo(
        sStt->getLastRtpTs(),
        sStt->getSPacketCount(),
        sStt->getSOctetCount(), 0,0) ));
    RtcpReportReceptionBlock *rb=new RtcpReportReceptionBlock();
#endif

    // Setting RTCP packet
    rb->set_rbssrc(this->ssrc_n);
    rb->set_flost(this->fractionLost);
    rb->set_cumlost(this->cumulativeLost);
    rb->set_seqhigh(this->max_seq);
    rb->setSeqHighCycle(this->cycles);
    if (this->jitter < 0) {
        rb->set_jitter(0);
    } else {
        rb->set_jitter((uint32_t) this->jitter);
    }
    rb->set_lsr(LSR); // Last SR timestamp
    rb->set_dlsr(DLSR); // Delay since last SR timestamp (received
from RtcpReceiver!!)
    sr->add_reception_block(rb);
    sr->set_sender_ssrc(sStt->getSSRC());
    sr->get_header().set_payload_type(200);

    return sr;
}
```

Listing 7: Method createSenderReport() in class RtcpSRManager

B.3 Configuration File “.minisip.conf”

```
<version>
    2
</version>
<account>
    <account_name>
        Franz Mayer
    </account_name>
    <sip_uri>
        sip:kth@iptel.org
    </sip_uri>
    <proxy_addr>
        iptel.org
    </proxy_addr>
    <register>
        yes
```

```
    </register>
    <proxy_port>
        5060
    </proxy_port>
    <proxy_username>
        kth@iptel.org
    </proxy_username>
    <proxy_password>
        password
    </proxy_password>
    <pstn_account>
        no
    </pstn_account>
    <default_account>
        yes
    </default_account>
</account>
<tcp_server>
    yes
</tcp_server>
<tls_server>
    no
</tls_server>
<secured>
    no
</secured>
<ka_type>
    psk
</ka_type>
<psk>
    Unspecified PSK
</psk>
<certificate>
</certificate>
<private_key>
</private_key>
<ca_certificate>
</ca_certificate>
<dh_enabled>
    no
</dh_enabled>
<psk_enabled>
    no
</psk_enabled>
<check_cert>
    yes
</check_cert>
<local_udp_port>
    5060
</local_udp_port>
<local_tcp_port>
    5060
</local_tcp_port>
<local_tls_port>
```

```
    5061
</local_tls_port>
<sound_device>
    dsound:test
</sound_device>
<mixer_type>
    spatial
</mixer_type>
<codec>
    speex
</codec>
<phonebook>
    file://C:\minisip_win32\Project\minisip\debug/.minisip.addr
</phonebook>
```

Listing 8: Sample configuration file (.minisip.conf)

Appendix C. Hands-On reference

C.1 Nistnet

1. Install *nistnet*[24] as described in http://snad.ncsl.nist.gov/nistnet/install.html
2. Configurate *nistnet*
   2.1. Load nistnet module with either of the following commands[25]:
      a) `insmod nistnet`
      b) `modprobe nistnet`
   2.2. Start nistnet with `cnistnet -u`
   2.3. Check if nistnet is running with `cnistnet -G`
3. Configure nistnet server (on Linux)
   3.1. Set each interface to the correct IP address:
      a) `ifconfig eth0 {C1,192.168.3.1}`
      b) `ifconfig eth2 {C2,192.168.2.2}`
   3.2. Set routing table
      a) Check route table with `route -n`
      b) `route add -net 192.168.3.0 netmask 255.255.255.0 gw 192.168.3.1 dev eth0`
      c) `route add -net 192.168.2.0 netmask 255.255.255.0 gw 192.168.2.2 dev eth2`
   3.3. Set IP forward flag: `echo "1" > /proc/sys/net/ipv4/ip_forward`

24 Note: nistnet is only available for Linux.

25 Note: usually `insmod` should work, but if nistnet is not listed by the command `lsmod`, try `modprobe`.

4. Configure Windows clients
    4.1. Set connection settings[26]
        a) IP address should be the ones defined in Figure 4.1 (A2 or B2 respectively)
        b) Subnet mask should be `255.255.255.0`
        c) There should be no entry in *Default Gateway* for the LAN network.
    4.2. Change routing tables with Windows Commands:
        a) Check routing table with `netstat -r` or `route print`
        b) On A2: `route add 192.168.3.0 mask 255.255.255.0 192.168.2.2 metric 2 if [`**`PCI Fast Ethernet Adapter`**`]`
        c) On B2: `route add 192.168.2.0 mask 255.255.255.0 192.168.3.1 metric 2 if [`**`PCI Fast Ethernet Adapter`**`]`
    4.3. Other important network issues
        a) It is **ESSENTIAL** that every other network interfaces is disabled (if you have two or more network interfaces). Otherwise minisip will create error message "`SipMessageTransport: sendmessage: exception thrown!`" and will not be able to send or receive any RTP packet!
        b) Check firewall for blocked ports, programs etc. To be sure that the firewall is not interfering turn firewall off.
5. Configure minisip
    5.1. Delete the entries of tags in *.minisip.conf*:
        a) `<sip_uri>`
        b) `<proxy_addr>`
6. Start minisip: `call test@{192.168.3.2,192.168.2.1}`

C.2 Programming Environment

This appendix describes how to set up a Visual Studio 8.0 project, as has been used in this thesis. To do so, it is necessary to have access to a working Visual Studio project, such as minisip on ccsser2 in the ccslab at KTH.

1. Needed files from folders (for the sake of compability files should be saved in same folders)
    1.1. C:\minisip_win32\
    1.2. C:\OpenSSL\
    1.3. C:\DX\Include\dsound.h
    1.4. C:\DX\Lib\x86\dsound.lib, dxguid.lib
    1.5. C:\boost_1_33_1\boost\
    1.6. C:\boost_1_33_1\bin\boost\libs\date_time\
2. Configure header and include path Tools > Options; Projects and Solutions > VC++

26 To be found in *Control Panel > Network Connections > Connection Interface > Properties > TCP/IP Internet Protocol.*

Directories

2.1. add include-paths

a) [Project-Folder]\minisip\include\

b) [Project-Folder]\lib*\include\

c) [OpenSSL-Folder]\include\

d) path to dsound.h

e) path to boost directory in C:\boost_1_33_1\

2.2. add library-paths

a) [Project-Folder]\Project\lib*\debug\

b) [OpenSSL-Folder]\lib\VC\static\

c) path to dsound.lib and dxguid.lib

d) path to boost directory:
C:\boost_1_33_1\bin\boost\libs\date_time\build\libboost_date_time.lib\vc-8_0\debug\threading-multi\

3. Building project

3.1. This has only be done, when there have been any changes. If minisip project has been copied from a working environment, e.g. ccsser2, step 4 can be skipped.

3.2. Rebuild (Build > Rebuild) the following projects in the right order; this have to be done only if the solution has not been built yet

3.3. Note: Rebuild does build the project from the scratch, whereas Build does only build the changes!

3.4. Note: You have to put the corresponding include path for each project at top2, e.g. compiling [Project-Folder]\Project\libmutil you have to put [Project-Folder]\libmutil\include at the top of all included files of minisip – these has to be done before starting to build each project!

3.5. Building order:

a) [Project-Folder]\Project\lib*

b) [Project-Folder]\Project\minisip

4. Configure minisip settings in file C:\minisip_win32\Project\minisip\debug\minisip.conf

5. System paths

5.1. check if you have set the system variable HOME[27] to C:\minisip_win32\Project\minisip\debug

5.2. and restart Visual Studio

6. Now you should be able to run it (Debug > Start (Without) Debugging)

27 Can be found in Control Panel > System > Environment Variables

C.3 Programs Used

The following programs has been used throughout this Master thesis:

- Writing Master thesis
  - OpenOffice for writing and diagrams
    http://www.openoffice.org/
  - EventStudio 2.5 for sequence diagrams
    http://www.eventhelix.com/EventStudio/
  - Poseidon for UML for class diagrams
    http://gentleware.com
  - Microsoft Office Visio Professional 2003 for further diagrams
    http://office.microsoft.com/
- Programming and Testing
  - Microsoft Visual Studio 2005
    http://msdn.microsoft.com/visualc/
  - minisip – open-source SIP soft phone
    http://www.minisip.org/
  - NIST Net – network emulator
    http://snad.ncsl.nist.gov/nistnet/
  - Ethereal Version 0.99.0 – network protocol analyzer
    http://www.ethereal.com/

COS/CCS 2006-14

www.kth.se