Diploma Thesis

on

REAL TIME SYSTEM DEVELOPMENT WITH UML: A CASE STUDY

By

Johannes Appenzeller

September 2003

Supervised by

Prof. A. Wegmann

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Communication Systems Division

School of Computer & Communication Sciences

Swiss Federal Institute of Technology

Lausanne

Produced under the Data Networks Architecture Group, University of Cape Town

Author:	Johannes Appenzeller Langmoosstr. 8 9442 Berneck Switzerland email: Johannes.Appenzeller@epfl.ch
Produced at:	University of Cape Town Department of Computer Science Data Networks Architecture Group Private Bag Rondebosch 7701 South Africa Tel: +27 21 650 2663
Local Supervisor:	Prof. Dr. P. Kritzinger email: psk@cs.uct.ac.za
Principal Supervisor:	Prof. A. Wegmann

Abstract

In this thesis we look at the challenges regarding VoIP and to the developer of an application providing this service. We explore CASE tools that can be used to model and verify the design of a VoIP application.

VoIP applications will not be accepted by the market unless it is able to provide an audio quality comparable to traditional phones. The voice module of the application that we analyse initially did not meet these requirements. We investigate how the design and implementation must be altered to meet them.

Although UML in its current specification is not adapted to the design of real-time applications, CASE tools exist that propose an extension of UML for this purpose. We investigate two of these - Rational Rose RT and Telelogic Tau - for their usefulness in re-engineering the application. We show their support partially covers our needs and we present novel UML concepts that would have been useful in resolving our task. We further demonstrate important new concepts of UML 2.0.

Acknowledgements

I would like to express my thanks and appreciation to all the people who made my internship with the DNA Group not only possible, but an amazing and unforgettable experience:

- Professor Alain Wegmann, for his guidance and supervision.
- Professor Pieter S. Kritzinger, for inviting me to join his research team and his co-supervision.
- Justin Kelleher for having provided the right input at the right moment.
- All the DNA fellows for their friendship, for being the source of a wealth of good memories and for introducing me to Cape Town and South Africa. And for proofreading my chapters.
- My family, and my friends in Switzerland, Cape Town and in other places of this globe for their love, inspiration and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In a society where broadband access to the Internet is becoming more and more common, real-time Internet services such as Voice over Internet Protocol (VoIP) - or Internet Telephony - are becoming available to the public. Delivering phone conversations at very low or even zero rate, this new technology has the potential to attract a large range of users. There is one barrier that needs to be overcome to be assured of all the public attention this new service deserves: the quality of the conversations needs to be comparable with traditional phone calls.

For their B.Sc. project at the Computer Science Department of the University of Cape Town, J. Landman, M. Marconi, M. Chetty, O. Ryndina developed a VoIP application named *ChattaBox* [1]. The final version of this application fulfilled the basic requirements regarding distribution, usability and sound quality: two persons could call each other, speak to each other, save their address books on a centralized server and leave voice messages.

Although these basic expectations were met, the major challenge - offering a voice quality comparable to traditional phones - was only partly solved. The application generated a good overall sound quality, but introduced a delay that made a fluent communication virtually impossible. In addition to that, the sound quality deteriorated over time.

The first part of this thesis concentrates on problem analysis. We investigate where the delay comes from and come up with propositions on how we can remove it. Interestingly, in order to minimize delay, we will need to look at a number of problems associated with bidirectional audio streaming over packet switched networks, which demonstrates that delay is one of the fundamental problems in this area. Our quest for a solution leads us into the world of real-time application programming and down to the operating system and hardware level.

The second part of this thesis concentrates on redesigning the voice module of ChattaBox using the Unified Modeling Language (UML). UML in its current definition (1.4) does not support concepts that help develop real-time applications. Nevertheless, there exists a number of tools that can support real-time development. All those tools extend UML with their own concepts to achieve this goal. The tools that particularly attracted our attention, and which we will examine, are Rational Rose Realtime and Telelogic Tau Developer Generation 2.

Apart from these tools, we study relevant new concepts of the emerging UML 2.0 standard and establish comparison with UML 1.x. We also give an overview of current research that is being conducted in the field of UML in a real-time environment, and introduce the UML Profile for Schedulability, Performance and Time.

1.1 Motivations and Objective

The primary objectives for this project are the following:

1. **Re-engineer the voice component of ChattaBox using UML, in order to**
    - **Decrease the voice lag, or voice delay, to a maximum value of 200ms.**
    - **Remove voice quality deterioration over time.**
    - **Target Application Environment: Large Local Area Network.**
2. **Investigate how UML and UML based CASE tools can be used to support the real-time application development.**

The motivation for this work is to improve ChattabBox, the telecommunications testbed of the Data Network Architectures Group of the University of Cape Town, and to grow our knowledge as well as the know-how of the DNA Group in the field of software engineering using UML, specifically UML in a real-time context.

1.2 Structure of Thesis

In **Chapter 2**, we *present ChattaBox* and make an assessment of its current functionality, bugs, deficiencies and enhancement ideas.

In **Chapter 3**, we *introduce VoIP*. After introducing the concept of "*voice quality*", we describe the issues and challenges related to VoIP, and demonstrate common techniques with which they can be addressed.

In **Chapter 4**, we *analyze* our*Problem Domain*. We detail how the voice module of ChattaBox is designed, point out reasons for its voice quality deficiencies, and discuss how we can address these problems. In the end, we present a set of general *design recommendations* for PC and WinXP based Voice over IP applications.

In **Chapter 5**, we look at *UML* in the context of distribution, concurrency and time. We compare relevant aspects of UML 1.x and UML 2.0 and discuss how they meet our modelling language requirements. We further introduce the UML Profile for Schedulability, Performance and Time, and give an overview of relevant research conducted in the field of real-time modelling with UML.

In **Chapter 6**, we examine and compare the two popular real-time *UML CASE tools* Rational RoseRT and Telelogic Tau Developer Generation 2.

In **Chapter 7**, we present our *design* and show how the activity diagram can be annotated to express real-time constraints.

In **Chapter 8**, we describe our *implementation*.

In **Chapter 9**, we discuss whether the project and application specific objectives have been met, describe how we managed this project, give a self-assessment and suggestions for future projects of this type before we make our final conclusion.

Chapter 2

Presentation of ChattaBox

In this section we describe the state of the art of ChattaBox in the beginning of this project. We establish a catalogue of points outlining current deficiencies and how overall quality and attractiveness of the application can be improved. From this catalogue we are going to establish the goals for this project.

2.1 Presentation

ChattaBox, the VoIP application under consideration, was designed and implemented by 4 Honours students from the Computer Science department of the University of Cape Town. It was a case study conducted to investigate software engineering of concurrent systems using the UML and the Specification and Description Language (SDL).

It is designed as a client-server application, and consists of three components. The *ChattaBox Client* is the interface to the user. It primarily allows two peers to communicate via voice. The *Orchestration Server* is primarily responsible for registering new users, making user data persistent and relaying calls between users and other orchestration servers. To balance the load of user data, it can be stored on one or more *Container Servers*. More features of the application are described in Table 1.

The signalling parts of the application are realized using the Session Initiation Protocol (SIP). Once a call has successfully been established, the Real-Time Protocol (RTP) is in charge for transmission of the voice data generated on both ends.

2.2 User Level Analysis

Although the application meets the basic requirements concerning performance, offered features, voice quality and robustness, it cannot compare - yet - with industry standard VoIP applications.

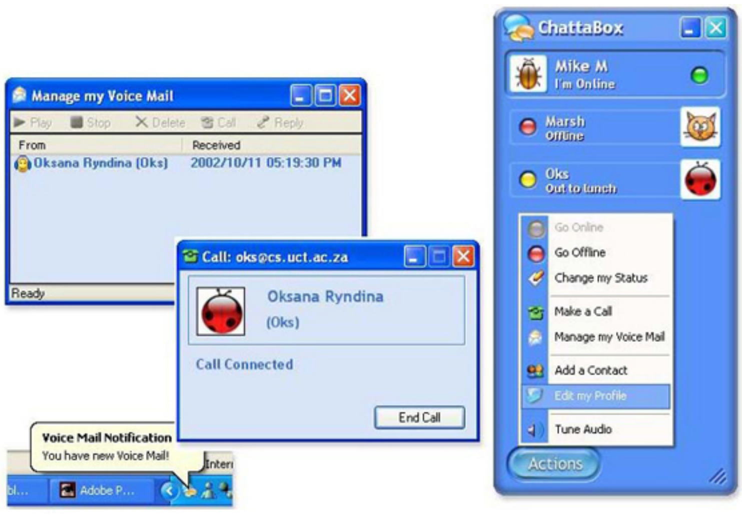


Figure 1: The client user interface of ChattaBox.

Below is a list of enhancement ideas (Tab.2), deficiencies (Tab.3) and bugs (Tab.4) that were found during User-Level Analysis. A resolution of these issues will greatly improve the quality, usability and overall attractiveness of the application.

The practical goals for this project were identified by addressing the issues by order of priority. The highest priority was given to resolve the deficiencies related to voice quality, specifically *end-to-end delay.*

Client Features	Server Features
• voice mail • list of contacts • possibility to manually change a clients status (online, offline, out of office, etc.) • user profile management • registration wizard for new users	• persistent client database • client data can be load balanced • secure user authentication

Table 1: Client and server features of ChattaBox.

Issue	Details
Conferecing	Allow users to set up a conference call.
Secure Calls	Allow encrypted calls.
Lower Bandwidth Consumption	Implement a compression algorithm or make use of a more sophisticated codec. For the moment, the PCM codec is used.
Call Switching	Allow users to handle more than one call by switching between them.
User switching	Allow to switch between user identities on the same client. This is necessary if different users on the same machine want to use the service.
Load balancing	The current implementation of the load balancing mechanism distributes the user data on several component servers. If the orchestration server needs to update this data,the complete user data file is loaded from the component server, updated and than sent back to the component server for persistence. This is expensive from both a computational and network point of view. An enhanced load balancing mechanism would *delegate* updating to the component server.
DBMS	Integrate a proper Data Base Management System, for efficient handling of user data.

Table 2: Enhancement possibilities for the initial version of ChattaBox.

Issue	Details
Voice Lag	Users often step into each others talk because of a voice lag between one and two seconds.
Voice Quality Deterioration	The sound quality of the received voice deteriorates for calls longer than 10 min.
Voice Quality Sensibility	Voice quality starts to suffer very quickly when the packet size is reduced and when the application is used on a network with several switches between the two communicating end points.

Table 3: Deficiencies with the initial version of ChattaBox.

Issue	Details
Client, List of Contacts	Cannot remove a contact when this contact is offline.
Client, Call Handling	If the window that permits to end a call is closed before pushing on the button ”End Call”, the call is not ended and a new call cannot be established.
Client, Popups	The small popups that show up when a call comes in or when a new user joins the service appear in irregular intervals with no apparent reason.
Client, Call Termination	The termination of calls often takes a considerable amount of time. Initiating a new call sometimes results in a crash of the client (doesn't respond anymore).
Orchestration Server or Container Server, Voice Mail	If a client deletes a voice mail, the mail appears again when the client software is restarted.
Client, Voice Mail Recording	The end of a voice mail is not smooth; some data seems to be repeated.
Orchestration Server	If the orchestration server encounters an error in its security certificate on startup, the server is shut down, but its process is not properly killed.

Table 4: Bugs in the initial version of ChattaBox.

Chapter 3

Voice over IP

In this chapter we introduce VoIP and the concept of voice quality. We describe the issues and challenges related to VoIP, and show techniques that can be used to address them.

3.1 Introduction

Internet Telephony, or Voice over IP (VoIP), is a rapidly emerging technology. Unlike the Public Switched Telephone Network (PSTN), which is used to place phone calls over circuit switched networks, VoIP uses the Internet Protocol (IP) to connect two callers.
IP Networks are packet switched and network bandwidth is shared by all users. As bandwidth is limited, the quality of a single network connection degrades when the number of its users increases. Unless the introduction of a mechanism for packet switched networks that allows users to obtain guarantees for a defined minimum network throughput[1], connection quality will remain a main issue in the VoIP world. VoIP applications have a need for timely delivery of data packets. This requirement necessitates a certain, known bandwidth. In an environment that was originally not designed for this purpose, such as the Internet or IP-Intranets, this minimal bandwidth is sometimes not available. This issue, and how to deal with it, is subject to research, and a number of solutions how to address this problem have been proposed.

[1]Currently, research is being conducted on how to efficiently guarantee a certain Quality of Service (QoS) to an application that communicates over a packet switched network. In the RFC1883, which describes the next generation of the IP Protocol, IPv6, QoS is proposed to be implemented on the IP level, by directly labelling the IP packets. The utilization of such an enhancement in connecting network devices like routers would obviously greatly simplify the environmental constraints on real-time services for IP networks. *Successful application of QoS requires the participation of* all *relaying network devices. As this is difficult to guarantee in heterogenous networks like the Internet, QoS will remain an issue in these networks even after introduction of IPv6.*

3.1.1 Why is VoIP an Important Technology?

VoIP today is mostly used to place long distance phone calls in order to save on telephone charges. Most people simply see in it a service that provides a mean to place free phone calls. Yet, its potential application areas are widespread. Some promising applications are mentioned in [5]:

1. *Toll-Free Corporate Telephony Services* - Toll-free intra-company voice and fax between corporate locations.
2. *Fax Over IP* - Toll-free or reduced rate fax-machine and fax between any two locations.
3. *PC-Phone to PC-Phone* - Toll-free voice between any two PC's on the Internet.
4. *IP-Based Public Phone Service* - New public phone services, at reduced rates (especially international calls), where voice is sent over the Internet or over new public IP networks.
5. *IP-Based Call-Centers* - A customer service web page that can connect the customer to an agent via the PC as a phone.
6. *IP Line Doubler* - A PC user at home or in a hotel with just one connection to the Internet could subscribe to a new service that enables a single phone line to carry one or more phone calls in addition to data.

3.1.2 Important Terminology

The following terms are going to be used throughout this report and deserve a short explanation.

Jitter is the variation of a digital signal from its ideal position in time. This variation can be due to packet delay in congested networks, modified routes, etc. Jitter is one of the fundamental problems of VoIP services. Its effects can be smoothed by creating an artificial delay through packet buffering at the receiver. A buffer with this function is called a *Jitter Buffer*.

End-to-end delay is defined as the time it takes for a signal generated at the speaker's mouth to reach the listener's ear. End-to-end delay is one of the major constraints in VoIP technology.

Codecs transform analog signals into digital bit streams and vice versa. For example, the G.711 codec is international standard for encoding telephone audio on an 64 kbps channel by pulse code modulating the analogue signal.
Software codec packages often do more than just coding the signal. The packages provided for VoIP, such as G.729, come with advanced features for speech transmission and audio processing, such as compression, voice activity detection (VAD) and error correction.

Service Quality	Sound Quality	Conversation Quality
• value added services such as voice mail • service availability • reliability • price	• loudness • distortion • noise • fading • crosstalk	All criteria for Sound Quality, plus • echo • end-to-end delay • silence suppression performance • echo canceller performance

Table 5: Details of Service, Sound and Conversation Quality.

Talkspurt A talker is said to be in talkspurt if he is uttering speech.

3.2 Voice Quality

In [6], on which this section is mainly based, Pracht et al. define Voice Quality (VQ) as follows:

> *"Voice Quality [...] is the qualitative and quantitative measure of the sound and conversation quality of a phone call."*

They argue that, on a very high level, Voice Quality (VQ) can be analysed from three main angles, namely *service quality*, *sound quality* and *conversation quality*. These three issues are detailed in Tab.5.

For the scope of this work, we are interested in understanding the *sound quality* and the *conversation quality*. Three elements emerge as the primary factors influencing these two issues in packet switched networks:

1. *clarity*,
2. *end-to-end delay* and
3. *echo*

We will now look more closely at each one of these points.

3.2.1 Clarity

Clarity can be described as *speech intelligibility*, indicating how much information one can extract from a conversation. The main factors that affect clarity are:

Packet loss is the main reason why clarity gets affected. Packet loss happens because of jitter and network congestion. If a packet experiences a large network delay, it may arrive too late for use in reconstructing the signal. On the other hand, if a network is congested, buffers in routers may overflow and start to drop packets. Because of delay constraints, retransmission of these packets cannot be applied.
Clarity suffers each time a packet gets lost, because a loss results in a data-gap within the playback stream. Some of the negative effects of such an interruption can be smoothed by applying some kind of loss concealment algorithm.
Packet size is an important factor that influences both cause and effect of packet loss. Larger packets are not only more likely to get lost. If they do get lost, they also cause more distortion on the playback voice stream than do small packets.

Compression affects clarity because the compressed signal is different from the original one. Depending on the compression algorithm used, this difference is more or less intelligible by the listener. Compression also affects end-to-end delay, as we will see later.

Voice activity detection (VAD) is used on the sender side. The goal is to avoid transmitting data containing silence. VAD reduces bandwidth between 35 - 50% [7].
Clarity can be affected because separating silence from voice is a difficult task due to the noise carried with each signal. In order to work properly, VAD needs to be configured manually to cope with the individual characteristics of the speaker's voice and the characteristics of the voice device. Even in optimal configuration, a voice activity detector may cut beginnings, endings or intermediary parts of speech.
With VAD, the receiver side needs to play substitutional data in silence periods. Options are complete silence or artificially generated noise that approximates the noise of the conversation. The later, known as Comfort Noise, is generally preferred by call-participants, as complete silence can give the disturbing impression of a dead line.
VAD is very important in conference calls. The accumulated noise of the (silent) conference participants can drown the speech-signal of the talker.

Quality of sound hardware The quality of the microphone, the speaker and the sound card as well as sampling rate (samples/sec) and sampling resolution (bits/sample) affect clarity.

3.2.2 End-To-End Delay

Delay is introduced by all devices on a signal's path, due to processing time or buffering.

Packet capture and **packetization delay** At the *sender* side, we find the packet capture delay and the packetization delay. The first is due to the time the recorder

Class	Delay	Influence on voice quality
1	up to 90ms	is not audible
2	90 - 150	acceptable
3	150 - 400	acceptable, but with degradation in voice quality
4	bigger 400	unacceptable

Table 6: Delay classification of the ITU recommendation G.114.

needs to fill up a packet of sound data before it gets sent over the network. This delay is a function of the packet size, the sampling rate and the sampling resolution. Packetization delay is the time that elapses from fetching a voice data packet from the sound device until it is pushed onto the network. It includes interprocess - or inter-thread - communication delay, compression delay and delay results from queuing in the socket.

Propagation delay and **network delay** is introduced by the *network* that the packet travels through. Propagation delay is the time it takes to vanquish the distance from end to end. This delay may be neglected for Local Area Networks, but not for long distances or satellite connections. The network delay is introduced by switches, routers, gateways, traffic shapers, and firewalls. Some devices add more latency than others; for example, a software firewall running on a slow PC adds more delay than a dedicated hardware-based firewall.

Jitter buffer and **depacketization delay** is found at the *receiver* side. The jitter buffer delay is a result of the size of the jitter buffer. The smaller the jitter buffer, the smaller the created delay, but the higher the risk to distort the voice playback by packet loss. The depacketization delay describes the time it takes a packet after arrival to be pushed into the jitter buffer and is mainly due to interprocess communication and decompression.

The ITU[2] recommendation G.114 suggests to classify delay in four categories, which are listed in Tab.6. In practice, delays around 200 ms are considered to provide acceptable voice quality. If a delay exceeds 250 ms, it becomes a major problem because talkers start to crosstalk, i.e. step into each others speech.

3.2.3 Echo

When a speaker can hear himself talking through the telephone speaker with a short delay introduced, we call this *echo*. Somewhere between the speaker and the listener,

[2]International Telecommunications Union, http://www.itu.int

the signal gets reflected.

Echo and delay are related in the sense that echo becomes an issue with round-trip delays bigger than 50ms. This delay is often widely exceeded by VoIP applications. Echo cancellers are a means to address this problem.

3.2.4 Summary

Figure 2 summarizes the different concepts from the previous sections and shows their relations. The most important information to be drawn from this figure is that the concepts of clarity, delay and echo are *interrelated.* It is thus impossible to address them separately. The most prominent example of this interrelation is the handling of jitter. As we can see in the figure, the jitter buffer size is related to clarity and delay. The reasons for this trade-off in is described in section 3.3.1.

The relations labelled <<discovered>> are shown here for completion; they were discovered during the analysis phase of the voice component. In the case of the VAD, we show later how it influences delay in a bigger and different way compared to what we mentioned in this chapter.

3.3 Maintaining Voice Quality

This section is a review on the most important methods and techniques that have been developed to address the problems of jitter, delay and packet loss. They differ in performance and complexity. For a comprehensive discussion, we refer the interested reader to [3], on which this section is mainly based.

3.3.1 Playout Strategies

In VoIP applications, the sender periodically generates sound packets as a result of the recording process. These packets are then sent through the network where they experience jitter, before they are received at the receiver side and played back to the listener.

The effects of jitter make uninterrupted playback one of the most challenging tasks in VoIP engineering. On the one hand a jitter buffer can delay the playback at the receiver to the beginning of a new talkspurt. This delay is called *jitter buffer delay*, and is determined by the size of the buffer. Ideally, this buffering-delay is big enough to guarantee that no packet is lost, i.e that the slowest packet is received before its playout is scheduled. It cannot be arbitrarily big due to end-to-end delay constraints.

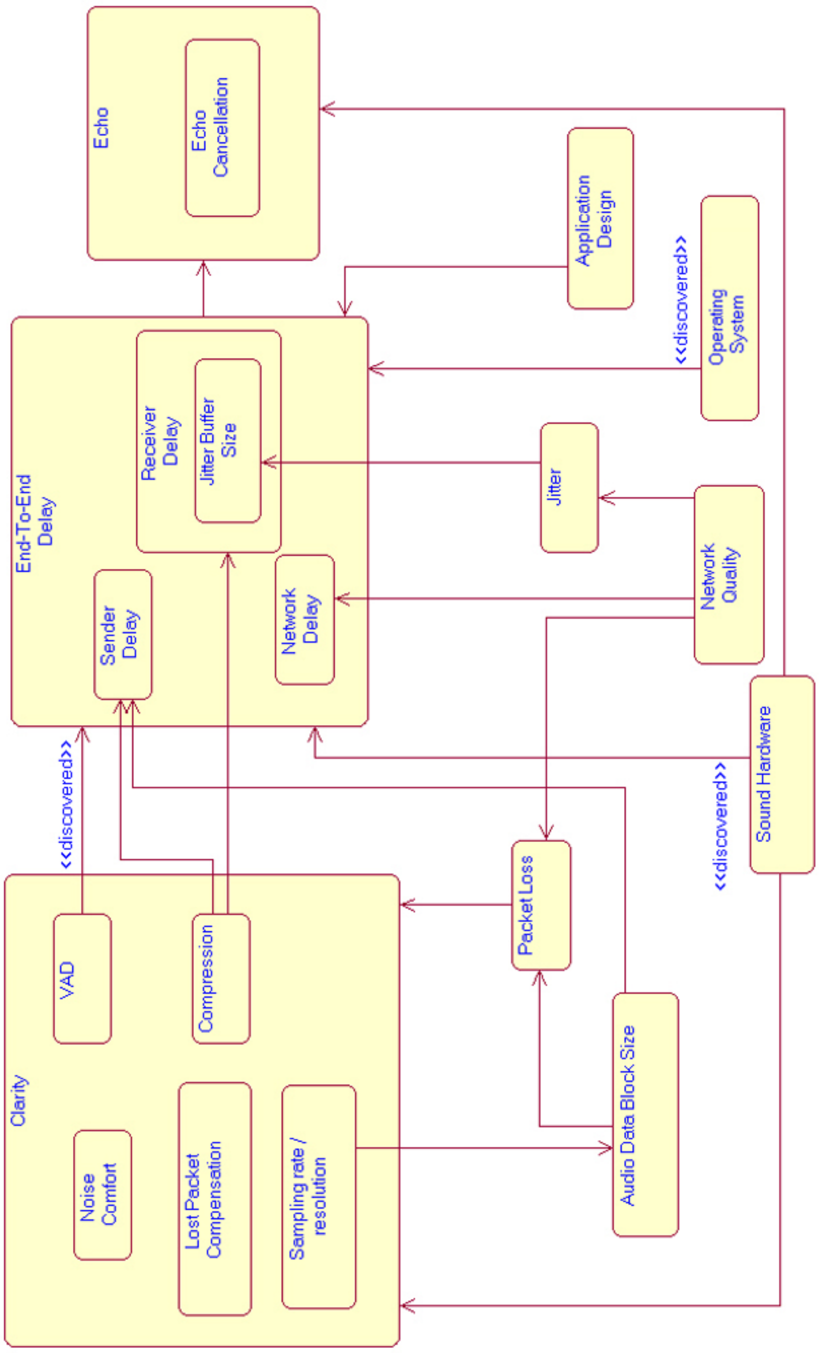


Figure 2: An illustration of the various dependencies affecting voice quality.

To find the right tradeoff between packet loss and delay, i.e to find the right jitter buffer size, is the job of a human or an *adaptive playout algorithm*, by implementing a certain *playout strategy*. The aim of such a strategy is to keep the delay small by allowing a fraction of packets to get lost due to late arrival.All playout algorithms are implemented at the receiver side.

Fixed Playout

In this strategy, the playout delay is fixed. It can be applied if the network delay is known and more or less constant throughout the conversation. Given the fact that network delay becomes very unstable as more networks are involved in relaying data-packets, this strategy is not well adapted for environments like the Internet.

Adaptive Playout

Adaptive playout strategies adapt to changing network conditions. They estimate the network delay and readjust the jitter buffer size appropriately. Most algorithms make use of the silence periods between individual talkspurts to readjust the jitter buffer since the extension or compression of a silence period has low impact on the voice quality. More recent algorithms not only propose readjustment between, but also readjustment within talkspurts, solving the problems of network delay variation within long talkspurts [4].

There are two main groups of adaptive playout algorithms: *reactive* algorithms and *predictive* algorithms. In the reactive approach, the algorithm measures jitter for the most recently received packets and acts appropriately. This approach is sensible to network spikes, i.e. long delays followed by the almost simultaneous arrival of a number of packets. Such spikes have influence the decision too much.
In the predictive approach, historical jitter measurements are not only used to adapt the buffer size but also to make short term predictions. This makes them less sensible to spikes.

3.3.2 Loss Concealment

Studies of network loss characteristics have shown that the vast majority of losses are single losses even when the loss percentage is as high as around 10% [2]. There exist a number of techniques to recover from or conceal single losses. They are either sender based or receiver based.

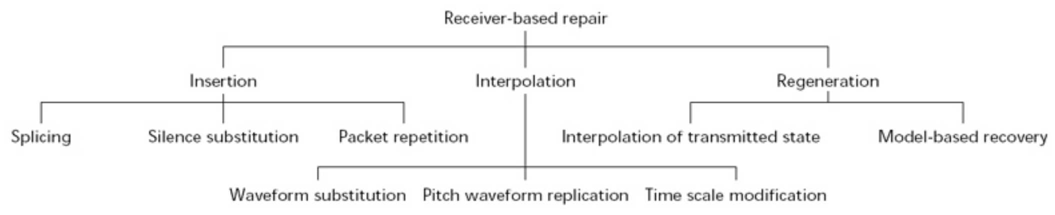


Figure 3: A taxonomy of error concealment techniques.

Sender Based Concealment Methods

Sender based concealment methods are based on active retransmission of the lost packet or channel coding techniques like Forward Error Correction (FEC).

Retransmission of the lost packet by the sender is an obvious means by which loss may be repaired.

Forward Error Correction is a concealment method where repair data is added to a media stream. Thanks to this information, packet loss can be repaired by the receiver of that stream with no further reference to the sender.

- *Media-independent FEC* is independent of the contents of the stream. These techniques add redundant data to a media stream, which is transmitted in separate packets.
- *Media-specific FEC* uses the knowledge of the stream data to provide more efficient repair methods. If units of media data are packets, it is logical to use the unit as the level of redundancy, and to send duplicate units. If a packet is lost, another packet containing the same data will be able to cover the loss. The first transmitted copy of audio data is referred to as *primary coding*, and subsequent transmissions as *secondary coding*. Secondary codings are usually encoded in lower quality than the primary to save on bandwidth.

The main disadvantage of retransmission is increased delay and bandwidth overhead. The same holds for media-independent FEC. They are therefore unappropriate for VoIP. Generally, media-specific FEC is recommended for VoIP.

Receiver Based Concealment Methods

Receiver based schemes do not substitute sender based schemes, but work in tandem with them. A sender based scheme is used to repair most losses; once the loss rate has been reduced this way, receiver based error concealment methods are an effective way of patching the remaining gaps in the data stream. Receiver based methods can be split into three groups: Insertion-, Interpolation- and Regeneration-based.

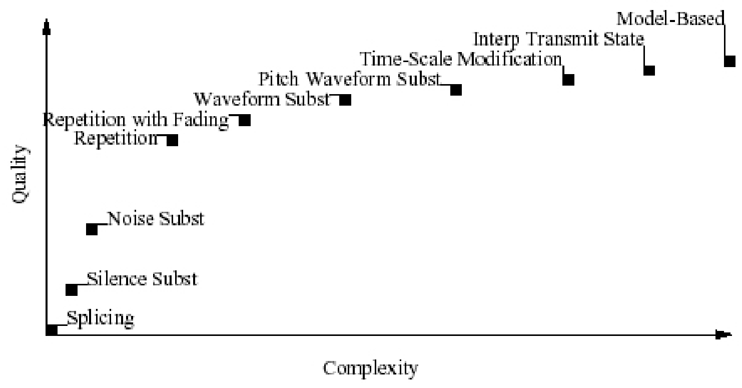


Figure 4: Rough quality/complexity trade-off for error concealment.

Insertion-based repair schemes derive a replacement for a lost packet by inserting a simple fill-in. The simplest case is *splicing*, where a zero-length fill-in is used; an alternative is *silence substitution*, where a fill-in with the duration of the lost packet is substituted to maintain the timing of the stream. Better results are obtained by using *noise substitution* or *repetition* of the previous packet.

Insertion-based repair techniques don't use the characteristics of the signal to aid reconstruction. This makes these methods simple to implement, but results in generally poor performance.

Interpolation-based repair schemes attempt to interpolate from packets surrounding a loss to produce a replacement for that lost packet. The advantage of interpolation-based schemes over insertion-based techniques is that they account for the changing characteristics of a signal.

Regeneration-based repair schemes use knowledge of the audio compression algorithm to derive codec parameters, such that audio in a lost packet can be synthesized. These techniques are necessarily codec-dependent but perform well because of the large amount of state information used in the repair. Typically, they are also somewhat computationally intensive.

An interesting measure is the quality/complexity trade-off between the different error concealment methods. This trade-off is shown in Fig.4.

3.4 Protocols Involved in VoIP

The protocols involved in VoIP need to provide two basic services required in telephony: signalling and voice streaming. For both services exist a number of proposed protocols and standards. A common choice is to use SIP for signalling and RTP for voice streaming. This is also the choice that was made by the designers of ChattaBox.

3.4.1 SIP

The Session Initiation Protocol, or SIP, is an application layer signaling protocol that defines initiation, modification and termination of interactive multimedia calls and conferences between users over Internet Protocol networks. Each session may include different types of data such as audio and video, although currently most of the SIP extensions address audio communication. As a traditional text based Internet protocol, it resembles the hypertext transfer protocol (HTTP) and simple mail transfer protocol (SMTP). It has been designed to be simple, scalable and easy to use, and is described in the Internet Engineering Task Force[3] (IETF) RFC 2543.

3.4.2 RTP

The Real-time Transport Protocol was developed by the Audio-Video Transport Working Group and has become an Internet standard. RTP is described in the IETF RFC 1889 specification as being a protocol providing end-to-end delivery services such as payload type identification, timestamping and sequence numbering, for data with real-time characteristics, e.g. interactive audio and video. It can be used over unicast or multicast networks. RTP itself however, does not provide all of the functionality required for the transport of data and therefore applications usually run it "on top" of a transport layer protocol. For applications that require minimal delays such as VoIP, UDP is often the preferred choice.

RTP usually works in conjunction with another protocol called the Real Time Control Protocol (RTCP), which provides minimal control over the delivery and quality of the data. It performs four main functions which are:

1. Feedback Information. This is used to check the quality of the data distribution. During an RTP session, RTCP control packets are periodically sent by each participant to all the other participants. These packets contain information such as the number of RTP packets sent, the number of packets lost etc., which the receiving application or any other third party program can use to monitor network problems. The application might then change the transmission rate of the RTP packets to help reduce any problems.
2. Transport-level identification. This is used to keep track of each of the participants in a session. It is also used to associate multiple data streams from a given participant in a set of related RTP sessions, e.g. the synchronisation of audio and

[3]http://www.ietf.org

video.

3. Transmission Interval Control. This ensures that the control traffic will not overwhelm network resources. Control traffic is limited to at most 5% of the overall session traffic.
4. Minimal Session Control. This is an optional function which can be used to convey a minimal amount of information to all session participants, e.g. to display the name of a new user joining an informal session.

Chapter 4

Problem Domain Analysis

In this chapter we focus on ChattaBox's voice component. After a short overview on audio streaming and sound programming in Windows, we make a thorough analysis of the component's design and give reasons for its deficiencies. We conclude by describing a set of design recommendations for PC based VoIP implementations.

4.1 A General Overview on Audio Streaming

To make this section accessible, we introduce a common design approach for audio streaming. The whole scenario is depicted in Fig.5. The concepts and data structures mentionned are listed and described in Tab.7.

At the recorder side, the audio signal is captured and encoded by the *recording device* (RD). The generated stream of audio data is cut into small *audio data blocks* (ADB) and associated with an *audio data descriptor* (ADD) waiting in a *recorder buffer*(RB). The RB is organized as a circular queue: The same ADDs are always filled, read, emptied and reinserted to the buffer. It usually contains more than one ADD to enable uninterrupted recording.
Once an ADD in the RB contains a full ADB, it is marked as such. The application, halting until this moment, takes the data and wraps it into an *network packet* before it is sent to its destination. The ADD that contained the data is cleared, marked as empty, and reinserted in the RB.

At the player side, a list of free ADDs wait to capture incoming ADBs from the network packets that arrive. To smooth jitter, the ADD is placed into a jitter buffer, where it remains until its playout time has arrived. If playout is scheduled, it is copied in the *playout buffer* (PB), which is organized as a multi-buffered circular queue. The

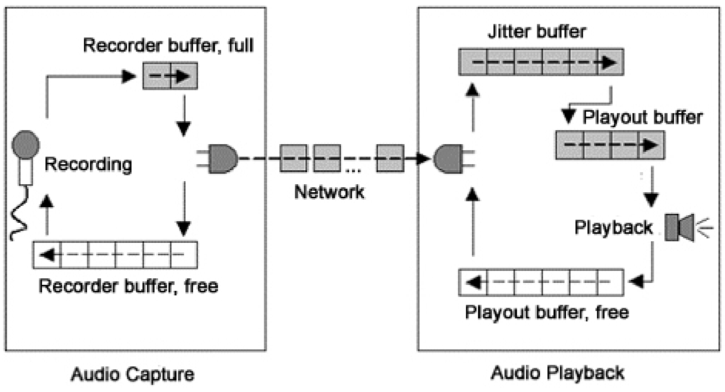


Figure 5: Streaming of voice, or audio in general, over a network.

Aside: Why is multiple buffering in the circular queues necessary?

Multiple buffering in the RB and the PB is necessary to allow smooth recording and playback. To understand why, consider the following example.
Scenario:

RB : size = 1 ADD
ADB : playtime = 1 sec
t_A : moment where the PD has marks the ADD as empty(played)
t_B : moment where the application reinserts the next, full, ADD into the PB.

The time $t_B - t_A$ can be very small but is *never* 0, because, after t_A, some processing must be done to reinsert the next full ADD into the PB. In this small gap of time, the PD is out of data which might result in a glitch. These glitches can be perceived by the listener.
If we have two buffers, the situation is different. While one ADD is being played in the background, the next ADD can be processed and reinserted into the RB. At t_A, a full ADD is already waiting, and playback can continue without interruption.A similar argument holds for the RD.
Buffers containing two ADDs are called to implement a *double-buffering* scheme; if more ADDs are involved, one speaks of *multiple buffering.*

playback device(PD) polls the PB for ADBs, sends the corresponding analog signal to the microphone, clears the ADD and reinserts it to the end of the PB.

4.1.1 The Windows Sound API

In windows, the sound device, or more exactly, the *driver* for the sound device, can be accessed through the Windows API for Sound. To analyse the voice component, we need to briefly discuss its main functionality.

Note that the windows terminology for ADD is *WaveHeader* (WH). The information describing the WH is contained in the fields WH.lpdata, WH.size and WH.flag among others. WH.lpdata is the pointer to the described ADB, WH.size describes its size and WH.flag its recording/playback status.

Term	Acronym	Description
Audio Data Block	ADB	Contains the pure audio data
Audio Data Descriptor	ADD	Structure containing the ADB, plus information how to interpret the ADB, such as sampling rate, mono/stereo, etc. In Windows, this structure is called a *WaveHeader*.
Network Packet	NP	The packet that is sent over the network. Contains the ADB, ADB interpretation information and routing information.
Recorder Buffer	RB	Circular queue containing at least two ADDs. The ADDs may be full (data recorded but not yet sent), or empty(ready to be filled with new recorded data). Full ADDs are being polled by the application in a FIFO manner.
Jitter Buffer	JB	After reception from the network, the APs are placed into this buffer, before they are scheduled to be played back.
Playout Buffer	PB	Circular queue containing at least two ADDs. Full ADDs are being played back in a FIFO manner.
Recording Device	RD	The device capturing voice and continuously associating a free ADD with new ADB.
Playout Device	PD	The device playing back audio data by continuously reading ADBs from ADDs.
List of Pointers	LP	A list of pointers, one for the RD and one for the PD, managed by the sound driver and referencing the available free ADDs in the RB respectively the full ADDs in the PB.

Table 7: Terms and Concepts in Audio Streaming.

Recording

The important operations related to recording are:

- `waveInOpen()`: Open and configure the RD. As a result of this operation, we dispose of a handle to the RD and a handle to an event that we will label `eventRecDone`. This event fires each time a WH has been successfully filled.
- `waveInStart()`: Start recording. The device driver has a list of pointers (LP) to empty WHs. If this list is empty, the data is thrown away without notifying the application level.
- `waveInPrepareHeader()`: Prepare a WH to contain a block of audio data.
- `waveInAddBuffer()`: Add a pointer to a prepared, empty WH to the LP of the RD. As soon as recording is started, the ADB of this WH is filled. Once the recording has successfully completed, `eventRecDone` is triggered and `WH.flag` is set to `recorded`.
- `waveInUnprepareHeader()`: Clean up the preparation performed by the `waveInPrepareHeader()` function. This function must be called *after* the device driver has succesfully filled a WH.
- `waveInStop()`: Stop all recording activity.

First, the RD is opened and started. Then, it is continuously provided with pointers to prepared, empty WHs; as soon as the WHs are known to contain a full ADB - either by receiving `eventRecDone` or by checking `WH.flag` - this data application, before the WH is reset (or "unprepared" in the "unprepared" in the Windows terminology). Eventually, the RD device is stopped.

Playback

The important operations related to playback are:

- `waveOutOpen()`: Open and configure the PD. As a result of this operation, we dispose of a handle to the PD and a handle to an event, `eventPlayDone`. This event is trigged each time a WH has been successfully played.
- `waveOutPrepareHeader()`: Prepare a WH for playback.
- `waveOutWrite()`: Add a pointer to a prepared, full WH to the LP. Playback is started when the first WH is inserted in the LP. The function is non-blocking: Once the pointer is added to the list, playback is done in the background. When when the ADB contained by the WH has finished playing, `eventPlayDone` is triggered and `WH.flag` is set to `played`.
- `waveOutUnprepareHeader()`: Clean up the preparation performed by the `waveOutPrepareHeader()` function. This function must be called *after* the device driver has succesfully played a WH.
- `waveOutStop()`: Stop all playback activity.

First, the RD is opened. Then, ADBs are continuously associated with unprepared, empty WHs in the playout buffer. A WH that contains data is prepared, and its pointer written to the sound device's LP. After having been played, the WH is unprepared. Eventually, the RD device is stopped. On the application level, the WHs are managed in a circular buffer for both recording and playback.

The scheduling scenario is depicted in Fig.6, where data is illustrated as hexagons and pointers as arrows.

4.2 Examination of the Voice Component

The voice component of ChattaBox has three classes: the RTP class, SoundPlayer class and SoundRecorder class. These classes provide the functionality for streaming voice over the network.

4.2.1 Voice Capturing - the SoundRecorder Class

The SoundRecorder class is basically a wrapper around the Win32 Sound API. It provides methods for starting and stopping the sound device, and to deliver recorded WHs to its

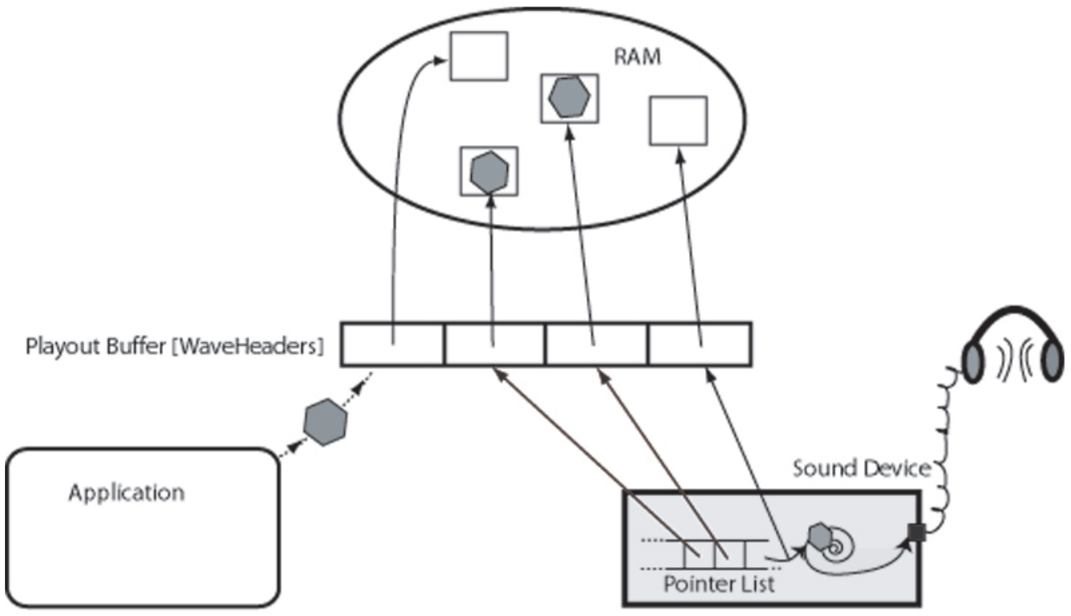


Figure 6: The playout scheduling scenario.

clients.

Class Configuration Parameters

Encoding Format:	Pulse Code Modulation (PCM)
Sampling Rate:	8000Hz
Sampling Resolution:	16 bit
Channels:	mono
Size of one ADB:	8000 bytes
Size of RB:	5 WH

First, the device is configured and started by calling the `startRecorder()` method. Then, ADBs are polled by repeated calls of `getSound()`.

Method startRecorder()

This method configures, opens and starts the RD.

```
WAVEHEADER recorderBuffer[];
int i;
SOUNDEVENT eventRecDone;

startRecorder(){
   waveInOpen(configuration, pointer to eventRecDone);
   for(i = 0 to recorderBuffer.size()){
       waveInPrepareHeader(recorderBuffer[i]);
       waveInAddBuffer(recorderBuffer[i]);
   }
   waveInStart();

   if(recorderBuffer[0] not filled){
       waitFor(eventRecDone);
   }
   return;
}
```

Listing 4.1: pseudo code for method startRecorder()

1. A series of WHs are prepared. The WHs are organized within the (circular) RB.
2. The RD is opened and configured. Pointers to the free WHs in the RB are added to the LP.
3. Recording is started.
4. The method returns after the first WH has been filled.

The RD fills the WHs described by its pointer list. Every time it finishes filling one WH, an `eventRecDone` is triggered the appropriate pointer is removed from its LP. If the LP is empty, the device discards the recorded data.

Method getSound()

This method is used to obtain recorded data from the RD.

```
WAVEHEADER recorderBuffer[];
int counter;
SOUNDEVENT eventRecDone;

getSound(){
   waitFor (eventRecDone);
   audioBlock = recorderBuffer[counter];

   recorderBuffer[counter].flag=0;
   waveInPrepareHeader(recorderBuffer[counter]);
   waveInAddBuffer(recorderBuffer[counter]);
   counter = (counter+1) mod recorderBuffer.size;

   return audioBlock;
}
```

Listing 4.2: pseudo code for method getSound()

1. The application waits until it gets notified (by `eventRecDone`) that a WH has been recorded.
2. The ADB of the recorded WH is copied in a temporary buffer.
3. The WH is cleared and its pointer reinserted to the pointer list of the device.
4. A counter is incremented to represent the next WH in the RB.
5. The obtained ADB is returned to the calling entity.

The circular buffer is implemented via a fix-sized array and a circular counter that is incremented according to the formula $counter = (counter + 1) mod(RB.size)$.

4.2.2 Voice Playback - the SoundPlayer Class

The design of the SoundPlayer is similar to the one of the SoundRecorder. It wraps around the Win32 sound API to provide an easy way for playing ADBs. The playback is organized in the `startPlayer()` and the `playRecSound(adb)` methods.

Class Configuration Parameters

Encoding Format:	Pulse Code Modulation (PCM)
Sampling Rate:	8000Hz
Sampling Resolution:	16 bit
Channels:	mono
Size of one ADB:	8000 bytes
Size of PB:	5 WH

Method startPlayer()

This method configures and opens the PD.

```
SOUNDCONFIGURATION configuration;\\
SOUNDEVENT eventPlayDone;

 startPlayer(){
    waveOutOpen(configuration, handle to eventPlayDone);
    return;
 }
```

Listing 4.3: Pseudo code for method startPlayer().

1. The PD is opened and configured. The PB, containing a certain number of WHs, is already initalized.

As soon as the PD receives the first ADD, it will start playing.

Method playRecSound(adb)

This method is used to provide the PD with blocks of audio data.

```
 WAVEHEADER  playoutBuffer[];
 WAVEHEADER* current;
 int counter;

playRecSound(adb){
    if(first call){
        counter = 0;
        current = playoutBuffer[counter];
    }else{
        current = playoutBuffer[counter];
        loop until(current.flag == DONE);
        waveOutUnprepareHeader(current);
    }
    current.data = data;
    waveOutPrepareHeader(current);
    waveOutWrite(current);
    counter = (counter+1) mod playoutBuffer.size;
}
```

Listing 4.4: Pseudo code for method playRecSound().

1. If this is not the first call, wait until the current WH of the PB has finished playing.
2. Clear the current WH.
3. Prepare the current WH to contain `adb`.
4. Add a pointer to the prepared, full WH to the LP of the PD.
5. Circularly increment the counter to point to the next WH.

4.2.3 Streaming Over the Network - the RTP Class

The RTP class is not designed as a class in the object oriented sense, but rather as a set of methods and data structures necessary to control and relay the incoming and outgoing voice data streams.

On a high level, it works as follows:

1. Setup a UDP connection to the other side.
2. Create a SoundRecorder and a SoundPlayer, and call their start() methods.
3. Repeat(infinitely):
   (a) If a packet is received, play it by calling `SoundPlayer.playRecSound()`.
   (b) Get the next recorded ADB from `SoundRecorder.getSound()`, wrap it into a network packet and send it.

The relevant functions provided by this class are `startRTPSession()`, `processIteration()` and `endRTPSession`.

Class Configuration Parameters

Receiver Timeout:	10 ms

Methods startRTPSession() and processIteration()

`processIteration()` is a sub-process of `startRTPSession`.

```
startRTPSession(fromPort, toAddress, toPort){
    rec  = new SoundRecorder();
    play = new SoundPlayer();
    setupUDPConnection(fromPort, toAddress, toPort);

    rec.startRecorder();
    play.startPlayer();

    while(true){
        processIteration();
    }
}
```

Listing 4.5: pseudo code for method startRTPSession()

```
processIteration(){
    waitUntil(RTP_packet_received or RECEIVER_TIMEOUT);

    if(RTP_packet_received){
        receivedData = extractAudioData(RTPPacket);
        play.PlayRecSound(receivedData);
    }

    sendData = rec.getSound();
    packet = createRTPPacket(sendData);
    send(packet);
}
```

Listing 4.6: pseudo code for method processIteration()

Method endRTPSession()

This procedure stops the SoundPlayer and the SoundRecorder by calling `play.stop()` and `rec.stop()`.

Miscellaneous Issues

The class implements a additional functionality that is not described above: An ADB can be split and sent with multiple RTP-Packets. On the receiver side, it is played out only once the complete ADB has been received.

Although the class defines the complete RTP-header, it uses only one field, the sequence number. It corresponds to the number of packets that has been sent. The Real Time Control Protocol (RTCP) is not implemented at all. Therefore, this class cannot be classified as a RTP library, but rather as a library for bidirectional real-time streaming.

Processor	Intel Celeron 1.7Ghz
Operating System	WindowsXP
Ram	256MB
Sound Device	SoundBlaster Live! 5.1 (PCI)

Table 8: Configuration of the main test system.

Processor	Intel Pentium II 299MHz
Operating System	WindowsXP
Ram	128MB
Sound Device	SoundBlaster AWE 16-bit (ISA)

Table 9: Configuration of the second test system.

4.3 Analysis of the Voice Component

In this section, we analyse it and identify the reasons for end-to-end delay and voice deterioration

Approach

The problem domain is split up in three sub-domains: *Sender*, *Network* and *Receiver*. Each sub-domain was analyzed for three issues: *timely processing*, *amount of buffered audio data* and *correctness of audio data*. Time was observed by generating timestamped logs using the windows `getSystemTime()` command. The obtained timestamp has a precision of 1ms. The amount of buffered audio data in the PB and the RB is observed using the windows SoundAPI methods `waveOutGetPosition()` for the PD and `waveInGetPosition()` for the RD. With the device driver, these methods return the amount of bytes played/recorded since the beginning of the session. By comparing this value to the total amount of data that was received or sent, we can determine the current buffer load with a precision of 1 byte. Correctness of data is observed by writing recorded data to a wave file just before being sent, and writing received data to a file just before being scheduled for play back. This is done to observe whether the data is correct (i.e. contains glitches, gaps etc.), without the additional constraint of time.

Note: The application does no extensive calculations such as compression or voice activity detection. Therefore, processing time is small (it has been measured to be under 1ms between blocking method calls) and can be neglected when analyzing delay.

Test Configuration

Our test configuration is made up of two system, the *main test system* and the *secondary test system*. The main system is used for tests on one machine; the second system is needed when analysis of communication between two machines is necessary. The two

Example: Causalities between blocking method calls.

If a `SoundPlayer.playRecSound()` waits for a new, empty WH, no recorded packet can be sent in the interim, even though there may be data ready for processing. Therefore a sender-delay is generated, which increases the end-to-end delay. This additional delay could cause a PB underrun at the other end of the communication.
The scenarios in Tab.10 give more examples how the receiving and sending parts of the application affect each other negatively. It is not supposed to be complete.

No.	Situation Peer 1	Situation Peer 2	Possible Consequence
1.	`playRecSound()` blocks because PB full → sending delayed	RB low	possible player buffer underrun at at peer 2, although ADBs are ready for sending at peer 1
2.	`playRecSound()` blocks because PB full → sending delayed	(uninvolved)	sender-delay increases, leads eventually to recorder buffer overrun at peer 1
3.	`getSound()` blocks → receiving delayed	(uninvolved)	player buffer underrun at peer 1 if player buffer level is low enough. This happens although packets might have already been received by the network device

Table 10: Causalities between blocking method calls and possible outcomes due to sequential application design.

machines are linked by an indirect 10Mbit link. Their configurations is given in the Tab.8 and 9.

4.3.1 Sequential Design and Waiting States

As can be seen from listing 4.6, receiving, playing, capturing and sending of audio packets is organized sequentially in a loop. This design is problematic when the ADB size is reduced. The SoundPlayer, SoundRecorder and the receiving part of the application sometimes have to wait for external events. As both playing and recording are controlled in sequence, a waiting state in one part creates causalities between sender-delay and receiver-delay that can be complicated to comprehend.

The voice component has three waiting states. All of them are necessary, but not all of them are implemented efficiently. We will discuss them and add identifiers to the statements for later reference. An identifier describes the *purpose* of a wait statement, not its current implementation.

1. *Class*: `RTP`
   Method: `processIteration()` , Listing 4.6
   Line 2: `waitUntil(RTP_packet_received or receiver_timeout)`

Purpose: Wait a certain amount of time for new packets, in order to provide them to the application as soon as received. If no packet is ready to be received from the network device, the application waits `receiver_timeout` sec.
Corresponding statement in real code: `select()`
Identifier: receiverWait

2. *Class*: `SoundPlayer`
   Method: `playRecSound()`, Listing 4.4
   Line 11: `loop until(current.flag == DONE)`
   Purpose: If the PB is full, wait until the next WH has finished playback. This can happen for several reasons, one is when the application enqueues WHs faster than the SoundPlayer can dequeue (play). For example, if a burst of 6 packets is received, each one containing 1 second of data, and the SoundPlayer can buffer 5 packets in its PB, the 6th packet must wait 1 second before it can be buffered.
   Corresponding statement in real code: `while(current.flag != 3)` Using a `while` loop to check the status of the WH is not efficient.
   Identifier: playerWait

3. *Class*: `SoundRecorder`
   Method: `getSound()`, Listing 4.2
   Line 6: `waitFor(DONE)`
   Purpose: Wait until the next packet is recorded.
   Corresponding statement in real code: `WaitForSingleObject(eventPlayDone)`
   Identifier: recorderWait

Thus, in one loop, the worst case scenario regarding end-to-end delay is equal to $2 * ADB_{playtime} + receiver\_timeout$!

Buffer overruns and underruns described in Tab.10 harm the voice quality. Buffer overruns at the recorder results in lost data; buffer underruns at the player produce glitches, which are negatively perceived by the listener. If the application is run in a small LAN environement, buffer-underruns happen rarely, as long as neither side is interrupted in its execution by another process running on the same machine. As soon as the application is confronted with bigger LANs and network congestion, buffer underruns happen regularly.

Conclusion 1 Overruns in the recorder buffer and underruns in the playout buffer must be prevented, since this has the most negative effects on voice clarity.

Conclusion 2 Organizing all tasks related to voice streaming in a sequential manner causes dependencies between the sender-delay and the receiver-delay. These interdependency can have a negative influence on the over-all performance, and favor buffer overruns/underruns.

4.3.2 Audio Data Block Size

An apparent reason for the big *end-to-end delay* is the configuration of the sound device. End-to-end delay is composed into sender-, receiver- and network-delay (see 2). One factor that influences the sender delay is the size of on audio data block. The amount of sound in seconds contained by one ADB is defined as follows:

$$ADB_{playtime} = \frac{ADB_{size}}{sampling_{resolution} sampling_{rate}}$$

By insertion of the configuration parameters for the player and recorder class, we obtain

$$\left.\begin{aligned} &ADB_{size} = 8000 bytes \\ &sampling_{rate} = 8000 samples/sec \\ &sampling_{resolution} = 16 bit/sample = 2 bytes/sample \end{aligned}\right\} ADB_{playtime} = 0.5 sec$$

Thus, with the above configuration, the audio stream is split up into blocks of 0.5 seconds. In other words, a packet capturing delay of 0.5 seconds is created! This delay is introduced by the sender only; network and receiver delays are yet to be added in order to obtain the complete end-to-end delay.
As we are operating the application on a LAN, the network delay is in the order of magnitudes smaller than the delay introduced by the sender (typically $<$ 1ms), and can be neglected on this stage of analysis.
On the receiver side, no jitter buffer is implemented; playback is started as soon as the first packet arrives, thus the only delay created is due to processing. Processing time has been measured to be smaller than 1ms, and can be neglected at this stage of analysis as well.

Conclusion 3 The size of the ADBs is the main source for end-to-end delay in the analysed application. It must be reduced to minimize the packet capturing delay. A common value for the size of one ADB is to contain data proportional to 20ms of playtime [2].

While experimenting with the ADB size, we observed that the smaller the size, the better was the delay, but the voice clarity got worse. Voice started to play back with a lot of interruptions and glitches, that made conversation virtually impossible. One reason was addressed in the previous section and is due to the sequential design of the application.

Say an ADB corresponds to 20ms of audio. Thus, in order to prevent a playout underrun, a packet must be scheduled for playout at least every 20ms. The class parameters for the SoundPlayer class define the *receive_timeout* to be 10ms. In the case where this timeout is exhausted, only 10 ms are left for scheduling the next packet. These 10 ms are sometimes occupied by other processes managed by the OS, which would lead to a playout underrun.

Conclusion 4 While waiting states are inevitable, the application must be designed in a way that such a state doesn't prevent other tasks of the component to continue to execute.In general, the over all waiting time of the application must be minimized.

Another reason related to time management in WinXP. Timers in WinXP are inaccurate, and therefore so are time-dependent event notifications. By specifing a receive_timeout of 10ms, this does not mean that the OS returns exactly after 10ms. On the main test machine, the measured value was often around 16ms, but could even be higher! This inaccuracy has severe effects on the sequentially designed voice component handling small ADBs, where waiting states influence each other and cannot be controlled separately. We will address this important issue in section 4.4.1.

4.3.3 Playout Buffer Size

The number of ADDs that can be contained in the PB is another factor that contributes to end-to-end delay. If the PB contains 5 filled ADDs, then a delay of $5 * ADB_{playtime}$ is produced. In other words, the latest ADD in the buffer contains voice that was recorded by the talker at least $5 * ADB_{playtime}$ seconds before it is going to be played.

In the initial design, the PB was *oversized.* The designers aim was that it always contains 2 full WHs (double buffering), but it offers space for 5 WHs. The result is that the PB starts to fill up in the course of time; it contains more and more full WHs, therefore introducing a significant player delay.

Conclusion 5 Full, waiting WHs in the PB contribute to end-to-end delay. WHs should be played out as soon as possible after they where buffered. The PB should not be oversized; it should not offer space for more WHs than are necessary for smooth sound playback.

4.3.4 Recorder Buffer Handling

The handling of the recorder buffer was identified to be a main source for *voice deterioration* over time. With the current design, data can possibly be accumulated in the recorder buffer, a situation the design doesn't allow to recover from. This behaviour eventually causes a RB *overflow.*

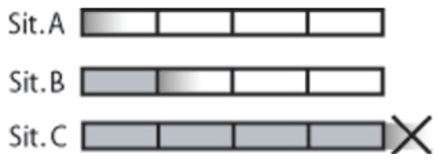


Figure 7: Situations in the recorder buffer overflow scenario.

Reconsider scenario 2 from Tab.10. It was said that, if the PB is full, sending is delayed. Assume that the RB contains one partially filled WH at position RB[0] and an empty WH and RB[1]. This is the optimal buffer situation (Fig.7, Sit. A). If the introduced delay is big enough so that the recorder starts recording RB[1] before RB[0] was sent, RB[0] will be sent only after RB[1] has finished recording. When the application calls `SoundRecorder.getSound()`, it will wait for `eventRecDone`, not considering the fact there is already a recorded ADB waiting, and that `eventRecDone` actually signals the successful recording of the *second* WH. Furthermore, the application will never know `eventRecDone` was lost. From this moment, `getSound()` will always be one packet behind. (Fig.7 , Sit.B)

By repeating the above scenario, it can easily be seen that the RB becomes completely filled. The RD, having no space for new recorded data, will reject data until a new empty WH becomes available. (Fig.7 , Sit.C) If the last WH was filled at time t_A and the next WH becomes available at time t_B, the data recorded between t_A and t_B is lost.

Conclusion 6 By not delivering all available, recorded ADBs in one access to the RB, the `getSound()` method generates sender delay and may eventually even cause a recorder buffer overflow which results in constant voice clarity deterioration.

4.3.5 Sensibility to Network Size

The smaller the ADB size, the more the application was found to be sensible to network conditions.

For example, for small[1] ADBs, if the two communicating computers are linked over a single hub network, voice quality is below the targeted quality but still quite good. If we keep this small ADB size and add two switches to the path between the communicating stations, the voice quality deteriorates to a degree where communication is nearly impossible.

This phenomenon is directly related to jitter. Switches increase the network delay; they read the packet header and relay it to its destination host. If the network load is high,

[1]Here, "small" means a size below 40ms of playtime.

this task can introduce varying delays. Combined with the problems due to sequential design of the application, the jitter can easily cause playout underruns.

Conclusion 7 Jitter, even though very small, is present in LANs and must be addressed. It can be smoothed by implementing a jitter buffer.

4.3.6 Session Termination

A session is terminated with the `endRTPSession()` method of the RTP class. Its call generates an unspecified behaviour, because it does *not* interrupt the infinite `processIteration()` loop that controls the communication behaviour before resetting behaviour before resetting the sound device. Two interfering methods of the same object are thus called in parallel: while the `processIteration()` loop, the thread created by the *stop* signal calls `endRTPSession()` which, in turn, calls `play.stop()`.

Unclean session termination was also an issue found during the user level analysis of ChattaBox. It is described in Tab.2.

Conclusion 8 Clean session termination should be introduced in order to obtain a stable voice component, that can be invoked repeatedly.

4.3.7 Protocol Implementation

RTP is only partially implemented, and RTCP is not implemented at all. The only RTP header field that is set is the sequence number. This number is equal to the number of packets sent. It is not used at the other side.

As the application runs in a controlled and static environment, it is not generally a problem and has no direct contributions to the end-to-end delay or voice deterioration. It must be kept in mind that if the application should be made Internet ready, or if conferencing is to be implemented, the information distributed by the RTCP Protocol is a necessary resource. Also, the fields proposed by the RTP Header should be used. For example, it is not possible for the voice component to communicate with a third-party RTP application, due to non-standard and missing header information.

4.4 Operating System and Hardware Level Issues

In this section we will analyse some issues that where resolved relatively late in the implementation phase. What was first thought to be implementation bugs was finally found to be operating system and driver dependent issues.

The first symptom was that audio would play with glitches, although the received data was correct and scheduled for playback on time - or so we thought. The second issue was that, for some reason, data would accumulate in the jitter buffer in the course of the conversation, therefore increasing end-to-end delay. We will investigate the reasons and solutions for these behaviours in the two following sections.

4.4.1 WindowsXP and Timers

Glitches, in our case, are most commonly the result of a of a buffer underrun at the PD. In our re-engineered SoundPlayer class, we implemented a double-buffering scheme (explained in section 4.1) to prevent buffer underruns. Double buffering was *guaranteed*: i.e. independent of whether new audio data was actually available, *some* data would *always* be buffered on time for the sound device to play. So why would there still be glitches?

Double buffering was thought to be the ideal buffering scheme in our application, because it is minimal in delay and leaves enough time for processing data. For scheduling of packets, we relied on the timely accuracy of the `eventPlayDone`, the event that notifies the application of successful playback. It was a major step towards sound quality improvement when we found out that this notification event is *not* accurate in time, due to how MicrosoftXP handles time and timers.

In their "Guidelines For Providing Multimedia Timer Support" [29], Microsoft gives some explanations in this regard:

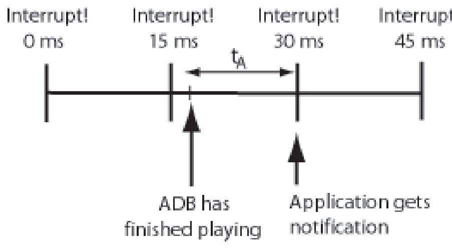
> Windows XP uses a periodic clock interrupt to keep track of time, trigger timer objects, and decrement thread quantum. When Windows XP boots, the typical default clock interrupt period is 10 milliseconds, although a period of 15 milliseconds is used on some systems. This means that every 10 milliseconds, the operating system receives an interrupt from the system timer hardware. When the clock interrupt fires, Windows performs two main actions; it updates the timer tick count if a full tick has elapsed, *and checks to see if a scheduled timer object has expired.* Timer objects are used by the system to track deadlines and to signal applications when a deadline is reached.

We found out that our sound device driver uses this clock interrupt to check whether an ADB has finished playing. If yes, it sets the `WH.flag` to played and triggers the `eventPlayDone`. This periodic handling of the clock has undesirable consequences on our application, because it introduces additional jitter.

The voice component periodically waits for event notification in three places: *recorderWait*, *receiverWait* and *playerWait* (see section 4.3.1). While the first two wait statements introduce a jitter that can be smoothed by the jitter buffer placed between the

Example: Jitter generated by a periodic operating system clock.

Suppose that the clock period is 15ms, and a packet has finished playing at time t. Then we will get the `eventPlayDone` with a delay of $t_A = 15 - t \bmod 15$. If the playout buffer is empty, the PD runs out of data for at least $t_A$ms, which produces a glitch in the audio playback.



receiver and the player, the jitter introduced by playerWait is not smoothed by default, because it is not expected. It can, if not taken into account, significantly reduce voice clarity, and introduce glitches.

To make matters worse: even if an event is to be *scheduled* every 15ms, it is not guaranteed that it *arrives* at the application level every 15ms. There is always some latency involved. Latency is in the majority of cases below one millisecond, but can sometimes have values up to several milliseconds. It can be minimized by making real-time processes or real-time threads run in a high priority class. But the only way to get rid of the jitter introduced by `playerWait` state is to have enough data buffered at the PD which will smooth the effects of the periodic clock and latency. Specifically, *double buffering is not enough for smooth playback on our test systems.*

Conclusion 9 Using WindowsXP as the operating system for this application increases the minimum achievable end-to-end delay. The real-time application designer needs to consider jitter introduced by wait statements and timers. The only way to address the effect of this jitter is to smooth it by using a jitter buffer.

Tab.11 shows qualitative experimental values obtained for the necessary amount of pre-buffered ADBs to obtain glitch free playback. It was established on our main testing system. Audio was played from a file using a nearly-finalized, reengineered version of the SoundPlayer class. The values in these tables suggest that the sound device on our main test system needs minimum size corresponding to 60 ms of audio data in order to be able to smooth jitter introduced by playerWait.

4.4.2 Recording and Playback Speed

The last major issue that had to be resolved before coming up with a working voice component was that during a conversation, end-to-end delay would steadily increase with

	Playout buffer size:			
ADB_{size}	2 ADBs	3 ADBs	4 ADBs	5ADBs
100 ($\hat{=}$ 12.5ms)	*1*	*1*	*3-4*	*5*
128 ($\hat{=}$ 16ms)	*1*	*2*	*5*	*5*
150 ($\hat{=}$ 18.75ms)	*2*	*3-4*	*5*	*5*
200 ($\hat{=}$ 25ms)	*3*	*5*	*5*	*5*
256 ($\hat{=}$ 32ms)	*4*	*5*	*5*	*5*

1=very bad quality, *5*=very good quality

Table 11: Qualitative measure of sound quality in function of the playout buffer size and the ADB size.

the course of time. This would happen while the voice clarity was perfect and no glitches where audible, which indicated that the playing device was playing data without any buffer underrun (which could be a source for increasing voice delay).
We identified a rather strange reason to be responsible for this increase in delay: The recording device recorded data faster than the playback device played it back! In other words, 1 min physical time is not the same as 1 min device time, and 1min recording time is shorter than 1 min playback time. Thus, the recording device produces data faster than the play device can consume. Because the voice stream is uninterrupted (if during the complete session 100kb of data is recorded, 100kb of data would be transmitted over the network and be played on the other side), this difference leads inevitably to an accumulation of data at the receiver side.

This problem was analysed and confirmed in two different ways. First, we observed, for a voice session, the amount of data that was recorded during 60s at the sender (the second test system) and the amount of data played during 60s at the receiver (the main test system). We used 8kHz sampling resolution with an 8bit sampling rate at both. The ADB size was 128b. This configuration values define that 480kb of data that should be produced on the recorder and consumed at the player within that time. In practice, the recorder produced around 479.8kb per 60s, whereas the player only consumed around 473.6 kb in the same time! The difference of 6.2kb corresponded exactly to the accumulated data at the receiver within the same time span. With 8kHz/8bit, 6.2kb of audio data takes 620ms to play. Therefore, with our configuration, the end-to-end delay is increased by around 1sec after every 90sec of conversation!
After this observation, we were interested in how the sound system behaves if we record and play sound on the same machine. We coupled the SoundRecorder and the SoundPlayer directly by a queue and observed the amount of data in that queue. The SoundRecorder and the SoundPlayer run on different threads and produced/consumed data as fast as they could, using 128b ADBs. Surprisingly, the result was the same! Although the difference was not that big. It took around 3min of running conversation to produce a 1s increase in delay.

Resolving this problem at its source (sound-device) was considered to be too time expensive and maybe even impossible. Two work-arounds where considered:

- Manually modify the sound data in order to compress data at the player if accumulation is detected. This could for example be achieved by making one ADB out of two by picking out every second sample of the ADBs.
- Allow the voice stream to interrupt by discarding silent packets at the recorder/sender. This would allow the jitter buffer at the receiver to naturally empty itself from time to time (normally, silence periods are detected several times per minute). This way, accumulated data would get played out, and delay would not get noticeable. Obviously, some replacement data would have to be played at the receiver during a silence period in order to prevent a playout underrun.

We favored the second idea, as it is not only a work around but a feature: Voice Activity Detection. VAD would also reduce bandwidth consumption up to 50% [6] [7].

Conclusion 10 The application designer cannot rely on accuracy of the system time, or in other words, (nearly) exact mapping of physical time to system time. Mechanisms need to be introduced to make the application indulgent to these constraints.

4.5 Design Considerations and Recommendations

In this section we will draw consequences from the above analysis and present a set of general design recommendations for voice streaming applications in WinXP. The parameters of the recommendations are tailored for our test system.

4.5.1 ADB and Playout Buffer Size

The size of one ADB is the most important factor for end-to-end delay. It should be kept as low as possible, but large enough to allow the application to do execute some activities while one ADB is playing or recording. A value between 20 and 40ms should be aimed.
The results obtained in section 4.4.1 show that the playout buffer should be chosen dependent on the ADB size.

Design recommendation 1 Based on the results showed in Tab.11, we suggest to use an ADB size of 16 ms (128 bytes) and a PB length of 4 buffers. With this configuration we approximate the lowest possible jitter buffer size that allows smooth, glitch free playback. It must be made sure that at each time and as soon as we get the `eventPlayDone`, *all* empty WHs in the PB are refilled and scheduled for playback.

Note: This recommendation holds for our main test system, and has shown to be applicable for our second test system. The difference in architecture of both systems suggest that the lower bound for audio buffering depends on the OS, the sound-device and the sound-driver used. By changing the OS, the sound device, its driver, or even by removing or adding extensions cards from or to the system (thus changing the number of devices that use interrupts), this bound might change.

Multi-Threading

How can the tasks of the voice component be organized in order optimize its performance? As pointed out in the previous section, organizing them in a sequential manner is *not* recommended; some tasks should definitely run in parallel.

Threads are the means to obtain parallel tasks execution within the scope of one process. But the handling of threads produces processing overhead, and inter-thread communication has negative effects on the performance of applications dealing with low time units (due to event notification) and tends to be a source for programming errors. One is therefore generally interested to keep the number of threads as low as possible.

To identify the threads for the new voice component, we established the following guidelines:

1. Each thread must have at least one waiting state.
2. The waiting states in one thread must not interfere.

The following tasks need to be addressed:

Main task	Sub tasks
Recording	Provide empty WHs to the sound device. Handle full WHs received from the sound device.
Sending	Process the data (e.g. compression, VAD). Pack it and send it over the network.
Receiving	Process received data (e.g. decompression). Perform error correction and jitter handling.
Playing	Periodically schedule one ADB for playout.

Precise, periodic scheduling of WHs to the PD has highest priority to avoid buffer underruns. These tasks must therefore run independently from the (waiting-)state of the rest of the application, and are to be organized within a their own thread.

The receiver part should also be organized within its own thread. This way, packets are processed as soon as the network device has received them. Another reason for running the receiver in a dedicated thread is that its tasks might be time consuming (e.g. decompression of data).

We can see that the tasks for recording and sending can be organized within one thread. Recording is accomplished in the background and doesn't consume any processor time except for pushing and polling WHs. To combine them, what we need to guarantee is that $ADB_{playtime}$ is bigger than the time needed for putting a packet on the network.

Design recommendation 2 Implement 3 threads: a *recorderSender thread*, a *receiver thread* and a *player thread*. The recorderSender thread runs independently from the other threads. The receiver thread and the player thread communicate. Each thread has one waiting state, as required: recorderWait, receiverWait and senderWait. As there is not more than one waiting state per thread, the waiting states cannot interfere with each other.

N.B.: If we could rely on precise time handling on an OS level, the recorder and sender thread could be combined with the playout thread. Playing and recording ADBs is done in the background, and (should) take the same amount of time. Sending a recorded packet takes only small processing time. The recorder and sender can thus run in the same thread as the playout thread. Although there are two wait states (one in `getSound()` and one in `playRecSound()`), only one of them is blocking: at the moment the player has played one ADB completely, the recorder has recorded one ADB completely as well, thus the application will not wait. Thus, the two wait states do not interfere.

4.5.2 Recorder Buffer

The size of the recorder buffer should be large enough to guarantee that no data is lost during recording. Consideration should be given to the amount of time the recording process can be interrupted by the OS. The RB should be indulgent to such an interruption.
If the RB is handled efficiently, its size does not contribute to end-to-end delay.

Design recommendation 3 We suggest a recorder buffer size of 20 ADBs. Waiting packets must be polled and sent as soon as they are available to the application. A packet should ideally be polled right after it got recorded.

4.5.3 Jitter Buffer

The current application doesn't implement a jitter buffer. An integration of the later is necessary, as it provides basic robustness against unpredictable delay-variations, be they the result of network influence or due to the absence of aperiodic event notifications.

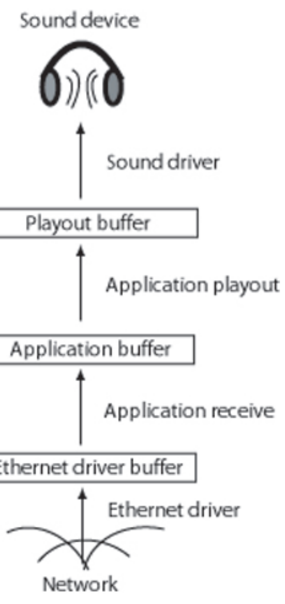


Figure 8: The tri-buffer situation at the player side of a voice communication.

Design recommendation 4 The primary jitter buffer should be placed between the receiver and the player part of the application. Its task is to smooth jitter caused by the sender and the network, and to reorganise packets if they arrive out of order. The playout buffer should be large enough to also have the role of a secondary jitter buffer, which smooths jitter caused by the SoundPlayer (application part, not device).

We will finally end up with a tri-buffer scheme, illustrated in Fig.8. It consists of an ethernet driver buffer, an application buffer, and a playout buffer that is used by the sound device driver. The main jitter buffer corresponds to the application buffer. The load in the ethernet driver buffer should always be kept at a minimum. The load in the application buffer is variable because of the jitter caused by the network and the sender. The load in the playout buffer is ideally constant and big enough to enable smooth playback.

4.5.4 Voice Activity Detection

Design recommendation 5 In order to keep delay low and voice quality high, Voice Activity Detection is an essential feature in our application and must be implemented.

Issue	Recommendation	Initial Application
Threads	3	1
ADB size	16ms	2000ms
Playout buffer size	4 ADBs	5 ADBs
Recorder buffer size	20	5 ADBs
Jitter buffer	implement	not implemented
Jitter buffer size	3	-
Voice Activity Detection	implement	not implemented

Table 12: Summary of design recommendations for the voice component.

4.5.5 Error Correction

Design recommendation 6 A mechanism should be implemented that creates a substitute for a lost packet. A good candidate is *repetition*, i.e. replace the lost packet with the last successfully received one (see Fig.4), because it is very low in complexity and processing time.

4.5.6 Summary of Recommendations and Comparison

In Tab.12, we give a summary of the established design recommendations and a comparison with the initial application.

4.6 Conclusion

As already discussed in chapter 3, finding the right trade-off between end-to-end delay and voice quality is the main issue for VoIP applications. This trade-off has consequences to all parts of the application and creates a lot of 'sub trade-offs'. Most of the design decision deal with that. The optimal solution for these parameters is determined by the applications requirements and the available environment. These requirements and constraints reach from the expectations of the user towards voice quality down to the type of operating system involved and the bandwidth available.

Our main goal must be to come up with a solution that has good voice clarity, i.e. plays without glitches and interruptions. We think that voice clarity is more important than minimal delay.

Our application deals with three types of delay: delay due to processing, delay due to physical transmission and delay due to queuing. In our environment, processing and transmission takes a very small amount of time. Thus, *our main focus must be to implement a solution that ideally manages buffered data in order to minimize waiting time.* Ideal queue management is our main way to minimize end-to-end delay, and can only be achieved through a multi-threaded design.

The delay margin of 200ms should leave us enough freedom to come up with a solution that satisfies both the requirements regarding voice clarity and end-to-end delay. In fact, if we follow the design recommendations outlined above, we should encounter a delay around 130ms if we run the application on a LAN. This value is obtained by summing the total buffered data in the jitter buffers/playout buffer ((3+4)*16ms = 112ms) and adding a 18ms extra delay for transmission, notification delays etc.

The following points describe issues that are not essential to obtain good voice quality in our target environment, but can further improve the component. Implementation is recommended if the target environment becomes the Internet.

- Integrate a codec specialized for VoIP, to further reduce network load.
- Make use of all RTP header fields and implement RTCP.
- Implement an adaptive jitter buffer.
- Implement a FEC algorithm.

Chapter 5

UML and Time

In the previous chapter we analyzed VoIP and environmental issues to find the design parameters for the voice component that was to be developed. After an introduction to real-time application design, we now investigate how we can use the Unified Modeling Language (UML) to model the structure and behaviour of the component. We assume that the reader is familiar with UML.

5.1 Motivation and Background

In recent years, real-time application design has become an increasingly important discipline of software systems engineering. With exploding computational power, these systems become more and more sophisticated. This sophistication creates a need for software design styles that manage this complexity and tools that enable the designers to test whether the system meets the (timely) requirements.

A real-time system is any system that responds and acts in a timely manner. Thus, "time" is a resource of fundamental concern, and activities must be scheduled and executed to meet their timely requirements. Timely requirements (and systems) are often classified into hard real-time, where failure to meet a deadline is treated as severe system failure, and soft real-time, where an occasional missed deadline may be tolerated [18]. Real-time system are often confronted with the need for concurrent execution of activities to fulfill these constraints. Furthermore, these concurrent activities may potentially need to run on several processing units.

Object oriented (OO) design, by incorporating successful software engineering patterns, has proved to be a successful approach to cope with system complexity. Maybe one of the most important properties to achieve this goal is that an abstract model of the problem domain can be mapped directly to the software. Encapsulation and inheritance further facilitate fault analysis and maintenance.

These advantages combined with a growing tool support made the OO approach increasingly popular within the real-time community. It was soon discovered that UML, being the de facto standard for OO modelling, can not be readily applied in the real-time domain.

Responding to the requirements of the industry, there has been extensive research in this area, and a number of companies released a tool with their own real-time framework, addressing the above challenges. For example, Rational with RoseRT, I-Logix with Rhapsody and Telelogic with its TAU 2 tools propose UML frameworks with real-time aspects.

5.2 Requirements for UML or a UML Based CASE Tool

Our requirements for UML or a UML based case tool are given as follows.

1. *Visualization and modelling of time and time constraints.* In our analysis of the ChattaBox voice component, we realized that the identified underlying time constraints were often interrelated. By modelling the application, we were trying to clarify this network of constraints, to simplify it and to find a minimal subset that fulfills our requirements.
   Note: The real-time constraints with ChattaBox are considered to be both soft and hard, e.g. packet delivery over the network (soft) and providing the playout device with sound data (hard).
2. *Modelling of concurrency and distribution* It has been shown that the application should run on more than one thread. We want to get a clearer picture of how these parallel tasks inter-operate and integrate with the rest of the system.
3. *Testing and model-verification* It should be possible to test the model during the design phase. In particular, we want to be able to validate time constraints.
4. *Code generation* Ideally, it should be possible to translate the whole model into code, to refine it by hand and to plug it into ChattaBox. Code generation relates to the previous point: testing and (visual) model-verification is often obtained through ”executing” the model, for which, in turn, some kind of executable code must be generated.

5.3 Reminder on UML

The UML is a standard language to visualize, specify, construct and document the artifacts of a software-intensive system[8]. It is the result of the merger of several early contributions to object-oriented methods. Since UML was standardized by the Object Management Group (OMG) [1] it has become the de facto standard modeling language for object oriented software engineering. In the mean-time it is widely supported by CASE tools. It is applied in a broad range of industries, ranging from health and finance to aerospace and e-commerce.

The UML facilitates communication between parties and improves understanding of the system that is in development. Different *views* provide different perspectives of the same system. Each view is made up of one or several *diagrams*. One diagram expresses a certain aspect of the system; none of them expresses the system as a whole.

5.3.1 UML Diagrams

This section provides a short reminder for the existing diagram types used in UML. It is mainly based on [8].

Diagrams are the most important artifacts that are created when modelling with UML. They express the structure, behaviour and architecture of the system. The designer typically chooses a subset of diagrams that best fits to his needs. Such a choice could be: use case diagram to model behaviour on a high level and capture requirements, class diagram to model the structure, sequence diagrams to model interaction between objects, and statechart diagrams to model state changes within the active objects of the application.

Structure is expressed in the following diagrams:

- The *class diagram* models the vocabulary and simple collaborations of a system. It illustrates the static design view. It shows classes, interfaces, collaborations and their relationships and associations. It is one of the central design diagrams.

- The *object diagram* is a special kind of class diagram. An object is an instance of a class. This essentially means that an object represents the state of a class at a given point of time while the system is running. The object diagram captures the state of different classes in the system and their relationships or associations at a given point of time.

Behaviour is expressed in the following diagrams:

[1]http://www.omg.org

- The *use case diagram* shows the interactions of the environment with the systems business processes. The environment is represented in the form of a set of *actors*, and the business processes as *use cases.* A use case is a visual representation of a distinct business process with observable results for the actors. The use case diagram shows which actors interact with each use case.
  Use cases and use case diagrams are created during requirements analysis phase. They are a major artifact to identify the vocabulary (e.g. classes) of a system.
- A *statechart diagram* models the behaviour of an individual object. It is a projection of the objects *state machine.* It specifies a sequence of states an object goes through during its lifetime in response to events, together with its responses to those events.
- The *activity diagram* is another projection of the objects state machine. Contrary to the flow from state to state as depicted in the statechart diagram, the activity diagram shows the flow of control from activity to activity in an objects life cycle. Activity diagrams are especially useful to model concurrent activities.
- A *sequence diagram* represents the interaction between different objects in the system. The important aspect of a sequence diagram is that it is time-ordered. This means that the exact sequence of the interactions between the objects is represented step by step. Different objects in the sequence diagram interact with each other by passing "messages".
- A *collaboration diagram* groups together the interactions between different objects. The interactions are listed as numbered interactions that help to trace the sequence of the interactions. The collaboration diagram helps to identify all the possible interactions that each object has with other objects. Collaboration and sequence diagrams are semantically equivalent.

Finally, **architecture** is expressed in the following diagrams:

- The *component diagram* shows a set of components and their relationships. Component diagrams are used to illustrate the static implementation view of a system. A component typically maps to one or more classes, interfaces or collaborations.
  Components are directly supported by a multitude of operating systems and programming languages. Enterprise Java Beans, COM+ components and executables are examples.

- The *deployment diagram* captures the distribution of the runtime elements of the application. It shows a set of nodes and their relationships. A node typically encloses one or more components.

5.4 How UML 1.x Fulfills Our Requirements

Here we will shortly review how UML 1.x is able to fulfill our requirements.

1. *Visualization and modelling of time and time constraints.*In [8] Booch et. al mention Time and Change Events. Events can be used in state machines to model the occurrence of a stimulus that can trigger a state transition. An event itself can take several forms, namely signals and calls. It can be triggered by the passage of time or a change in state.

   Time constraints are visualized as any other constraint. *Timing marks* are used to relate the constraint to a certain action. Timing marks are obtained by naming messages in an interaction. These messages are normally not named, but rather rendered with the name of a signal or a call. Given the timing mark, you can refer to any of three attributes of that message - startTime, stopTime, and executionTime.

2. *Modelling of concurrency and distribution.* A best practice solution for handling concurrency and distribution is the *active object.* Active objects are instances of active classes. These classes share the same properties as all other classes. An active object models an independent, encapsulated flow of control and is the abstraction of a process or a thread that can initiate control activity. Active objects can not only communicate by exchanging synchronous, but also by exchanging asynchronous signals and calls of operations, which allow them to continue their current activity while expecting a response.

   Another way of modelling concurrency is to make use of state-machines with *concurrent substates* that would execute in parallel. On an OS level, parallel substates translate to the same concepts as active objects, namely threads.

   Parallel activities are best visualized with *activity diagrams.*

3. *Testing and model-verification.* Testing and model-verification is not directly supported by UML.

4. *Code generation.* For code generation, a semantics for actions is necessary. UML only got action semantics in its version 1.5, which was adopted in March 2003. Before that, code generation was supported by an UML Profile called xUML, which basically defines a executable subset of UML and proposes its own action semantics and even its own action language, named Action Specification Language.

5.5 UML Profile for Schedulability, Performance and Time

Shortcomings specific to a certain application domain are usually addressed in *UML Profiles.* A UML Profiles is essentially a specialization of UML for a particular purpose. The *UML Profile for Schedulability, Performance and Time* is such a specialization. It was adopted by the OMG in March 2002, after having emerged from the need of the real-time industry for better UML support for real-time modelling and analysis. It primarily specifies and reduces semantic variations for the notions of time, time constraints and resources within UML 1.x.

Interestingly, this profile does *not* extend the UML with new architectural concepts. It rather introduces a set of annotations that can be used with the existing concepts and that tailor UML to model and analyse real-time systems. The profile focuses on properties that are related to modeling of time and time-related aspects such as the key characteristics of timeliness, performance, and schedulability. The ability to predict these characteristics based on analyzing models of software - including notably models that are constructed prior to a line of code being written is a fundamental objective of the specification

The profile is very general in order to adapt to any possible real-time framework. Tool vendors are encouraged to use the provided annotations as the basic resource of information for their own (proprietary) analysis methods.

In our opinion, the profile gives a good basis for future tool support for real-time modelling and, more importantly, analysis. It also underlines the importance that the contributing members of the OMG give to an UML that meets the requirements of the real-time world. If tool vendors really do adopt the defined vocabulary, then the developer of real-time applications is herewith given a notational language that will enable the developer to choose between a variety of tools for model-checking and analysis. This could potentially greatly improve and simplify development of real-time systems.

At the time of this writing, the profile exists essentially on paper. As we will see later, the upcoming standard of UML, UML 2.0, will introduce new time-related constructs; it is therefore very likely that this profile needs to be revised in the future to be applicable for the new standard.

5.6 UML 2.0 - Overview and Comparison

Based on the experience in modelling with UML and changes in software engineering best practices (e.g. the emergence of component based software engineering), the versions 1.x of UML showed several shortcomings. Therefore, the OMG sent out 4 Request for Proposels (RfP). The *UML Infrastructure RfP* requests a re-architecture of the UML to simplify its foundation and make UML a more modular and extensible language. The *UML Superstructure RfP* requests a number of enhancements to UML, which are directly usable to the UML modeller. The requirements originate from UML users that found UML unable to express their modelling needs. The *UML Object Constraint Language RfP* requests an explicit integration at the metamodel level with the rest of UML (both superstructure and infrastructure). Finally, the *Diagram Interchange* RfP seeks proposals to add sufficient detail of graphical and diagram information necessary to represent and interchange the diagrammatic aspects of UML models in an interoperable manner. The result of the proposals is in the meantime well-known as *UML 2.0*. This complete revision of UML is expected to be released as an official standard in the end of 2004 [20]. The Final Adopted Specification for the UML 2.0 Superstructure Specification was published on the 3rd of August 2003.

5.6.1 UML 2.0 vs. UML 1.x: Overview of Enhancements

The major enhancements in UML 2.0 in respect to UML 1.x are[39]:

- A first-class extension mechanism allows modelers to add their own metaclasses, making it easier to define new UML Profiles and to extend modeling to new application areas.
- Built-in support for component-based development to ease modeling of applications realized in Enterprise JavaBeans, CORBA components or COM+.
- Support for run-time architectures allows modeling of object and data flow among different parts of a system. Support for executable models improved in general.
- More accurate and precise representation of relationships improves modeling of inheritance, composition and aggregation, and state machines.
- Better behavioural modeling improves support for encapsulation and scalability, removes restrictions on mapping of activity graphs to state machines, and improves Sequence diagram structure.
- Overall improvements to the language simplifies syntax and semantics, and better organizes its overall structure.

We are now going to present some important points relevant for the UML *user* in more detail. They are specified in the Final Adopted Specification for the UML 2.0 Superstructure [10], which is often cited in the following.

5.6.2 Architectural Modelling with UML 2.0

One of the important new features of UML 2.0 is its improved support for modelling architecture i.e. modelling systems built of components and visualizing run-time architecture. In UML 1.x, the concept of Component is included but rather weak. Component models and architectural frameworks, such as Enterprise JavaBeans, CORBA Component Model and COM+ cannot be modelled easily. This is mainly because of four reasons: the specification of plug-substitutability of components is not possible, the notion of Interface is weak (especially missing are the possibilities to describe interactions between interfaces, the ability to specify outgoing signals and messages, and the possibility to define complex transactions), the specification of the requirements a component places on its environment is not possible, and because complex frameworks and patterns are difficult to express [28].

The new architectural concepts that are found in UML 2.0 are inherited from existing architectural description languages (ADL), such as the Rational RealTime Profile (see section6.1), ACME[2] and SDL[3]. We will now give a list of the new concepts and compare them to UML 1.x.

Component

In UML 2.0, a component is a self contained unit that encapsulates the *state* and *behaviour* of a number of classifiers[4]. A component specifies a formal contract of the services that it provides to its clients and those that it requires from other components or services in the system in terms of its *provided* and *required interfaces.* A component is a substitutable unit that can be replaced at design time or run-time by a component that offers that offers equivalent functionality based on compatibility of its interfaces. As long as the environment obeys the constraints expressed by the provided and required interfaces of a component, it will be able to interact with this environment. Similarly, a system can be extended by adding new component types that add new functionality. [22]

The component has multiple interaction points called *ports.* Each port is dedicated to a specific purpose and presents the interface appropriate to that purpose. The obligations that may be associated with an interface are in the form of various kinds of constraints (such as pre- and post-conditions) or protocol specifications in order to define complex transactions.

[2]http://www-2.cs.cmu.edu/ acme/

[3]ITU-T standard Z.100

[4]A *classifier* is a collection of instances that have something in common. A classifier can have features that characterize its instances. Classifiers include interfaces, classes, datatypes, and components. A *class* is a classifier that describes of a set of objects that share the same specifications of features, constraints, and semantics.**??**

With components, the architecture of a large-scale system can be more easily described by hierarchically decomposing its internal structure. This internal structure describes how the parts of a system interact to carry out the functionality of the whole, i.e. how the parts are encapsulated in higher-level parts, how these parts are interconnected, and what communication between parts is possible.

Changes made to UML 1.x: The component model has made a number of implicit concepts from the UML 1.x model explicit, and made the concept more applicable throughout the modeling life cycle. While in UML 1.x, the Component construct is merely a set of classes and doesn't have a state, the Component construct UML 2.0 is composed of parts (sub-components, state-machine, classes), ports and connectors and therefore gains their capabilities.

In UML 1.x, decomposition can be described in several ways, but all of them are insufficient if large systems are modelled. Decomposition can be achieved via the stereotype *Subsystem*. Compared to components, subsystems can be associated, but not connected - the links are between instances of classes within the subsystem. Subsystems do not support encapsulation-the interfaces are really interfaces of objects within the subsystem. It is not possible to identify subsystems, so it is not possible to interact with a subsystem without knowing some objects within the subsystem. On the other hand, class composition of UML 1.x does not allow the expression of relationships that only exist in the context of a particular whole, and the collaboration model falls short because it does not define a structure that may be instantiated - it is a way of describing how the parts, once instantiated, might interact [13].

Connector

Connectors either link ports of different components, or a port of a component with its internal parts (e.g. its state machine).
Changes made to UML 1.x: Connector is not defined in UML 1.x.

Port

Ports are the boundary that help separate different, possibly concurrent interactions. They fully isolate a classifier's internals from its environment. Ports therefore allow a classifier to be defined independently of its environment, making that classifier reusable in any environment that conforms to the interaction constraints imposed by its ports. The interfaces associated with a port specify the nature of the interactions that may occur over a port.
Changes made to UML 1.x: Port does not exist in UML 1.x.

Interface

An interface declares set of public features and obligations of a class. It can specify behavioural constraints on its features using a *protocol state machine* (A protocol state machine specifies the allowed call sequences on a classifier's operations.)
*Changes made to UML 1.x:*Interfaces can own a protocol state machine

Composite Structure Diagram

Architecture of a component is made explicit in the "composite structure diagram". This diagram shows the runtime architecture of a component, namely the sub-component roles, ports and their wiring through connectors.
Changes made to UML 1.x: This diagram type has been added.

5.6.3 Other Improvements in the UML Superstructure

Beside the third draft of the UML 2.0 Superstructure[11], Bran Selic's very informative tutorial on UML 2.0(and MDA) [20] is referred throughout this subsection.

Time and Timing Diagram

This section is inspired by [34]. A new diagram has been added to the UML 2.0 diagram family, the *timing diagram.* It is basically an interaction diagram that shows the change in state or condition of a lifeline - representing a Classifier Instance or Classifier Role - over linear time. The most common usage is to show the change in state of an object over time in response to accepted events or stimuli.

Of particular interest are also the changes made to the SimpleTime sub-package. The following quote gives an idea on what time-precision is to be expected:

> The SimpleTime subpackage of the Common Behaviour package adds meta-classes to represent time and durations, as well as actions to observe the passing of time. The simple model of time described here is intended as an approximation for situations where the more complex aspects of time and time measurement can safely be ignored. For example, this model does not account for the relativistic effects that occur in many distributed systems, or the effects resulting from imperfect clocks with finite resolution, overflows, drift, skew, etc. It is assumed that applications for which such characteristics are relevant will use a more *sophisticated model of time* provided by an *appropriate profile*[11].

The meta-classes that have been added are:

Duration	A duration defines a value specification that specifies the temporal distance between two time expressions that specify time instants.
DurationConstraint	A DurationConstraint defines a Constraint that refers to a DurationInterval.
DurationInterval	A DurationInterval defines the range between two Durations.
DurationObservationAction	A DurationObservationAction defines an action that observes duration in time.
IntervalConstraint	A IntervalConstraint defines a Constraint that refers to an Interval
TimeConstraint	A TimeConstraint defines a Constraint that refers to a TimeInterval.
TimeExpression	A Time Expression defines a value specification that represent a time value.
TimeInterval	A TimeInterval defines the range between two Time-Expressions.
TimeObservationAction	A TimeObservationAction defines an action that observes the current point in time.

Action Semantics

An important novelty in UML 2.0 is that it provides a semantics for actions. The defined actions are can be used to model fine grained behaviour on the level of traditional programming languages. An action takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty. In addition, some actions modify the state of the system in which the action executes. The values that are the inputs to an action may be obtained from the results of other actions using the activity flow model, or they may be described by value specifications. The outputs of the action may be provided as inputs to other actions using the activity flow model. Actions are contained in activities, which provide their context. They group into:

Communication actions	send, call, receive,...
Primitive function action	such as for arithmetic functions
Object actions	create, destroy, reclassify, start, ...
Structural feature actions	read, write, clear, ...
Link actions	create, destroy,read,write,...
Variable actions	read, write, clear,...
Exception action	raise

Improved Activity Graphs

UML 1.x defines an Activity Graph (AG) as a "special case of a state machine that defines a computational process in terms of the control-flow and object-flow among its constituent actions." This has led to much confusion and to many undesired restrictions on activity diagrams due to the mapping to state machines.
In UML 2.0, these restrictions have been removed. AGs have now a Petri Net like foundation, which enables unstructured graphs, graphs with "go-to's", and richer models of concurrency.
An important change is the new semantics of the term *Activity.* In UML 1.x, an activity was defined as a single step in the AG, while in UML 2.0 the Activity is the whole Diagram/Model, composed of several *Actions.* A single step is called *Action Execution.* Among other novelties, Activity can now have input- and output parameters, can respond to Interrupts and can handle Exceptions.
Actions within an activity communicate through *tokens*, which represent objects, data or loci of control. Tokens are offered by actions when their execution is complete, and form the input to new actions. They are the items that "flow" through the AG. They can queue up in the input and the output of an action and can get backed up in the network by a store. There are two flows in the AG, the *object flow* that relays data and objects from action to action, and the *control flow* that relays control messages.
The concept of *swimlane* has been extended and offers more partitioning options. It is called now *ActivityPartition*

Improved Sequence dDiagrams

Sequence Diagrams can be structurally decomposed, and different sequence fragments can be defined. Fragments can be:

Alternatives	which is a choice of behaviours; at most one will execute. Corresponds to an if-else statement.
Loop	describes a looping sequence
Option	is a special case of alternative
Break	represents an alternative that is executed instead of the remainder of the fragment (like a break in a loop
Parallel	describes concurrent sub-scenarios
Critical Region	describes a sequence that cannot be interleaved with events on any of the participating lifelines
Negative	identifies sequences that must *not* occur
Assertion	corresponds to the only valid continuation

5.6.4 How UML 2 Fulfills Our Requirements

1. *Visualization and modelling of time and time constraints.* Additionally to the time features mentioned for UML 1.x, UML 2.0 has new time-related constructs directly embedded in its meta-model (like timeConstraint) and a *timing diagram.*
2. *Modelling of concurrency and distribution.* Concurrency and distribution is very well supported through the new Component construct and the refined Activity semantics.
3. *Testing and model-verification.* In June 2001, the OMG sent out a Request For Proposal on an UML 2.0 Testing Profile . According to the OMG, the UML Testing Profile defines a language for designing, visualizing, specifying, analyzing, constructing and documenting the artifacts of test systems. This RfP is still in an early status. A good discussion on the profile and it's alignment with TTCN-3, which is the currently most widely adopted testing language, can be found in [33].
4. *Code generation.* Semantic variation points of its modeling language are reduced, and it defines an action semantics. UML 2.0 is going to be well adapted for code generation.

5.7 Overview of Relevant Research

In this section, we will give a brief overview on research regarding the application of UML in the real-time domain. A common pattern that was found is that annotated UML-Diagrams, especially the UML-Statechart diagram, are translated into a semantically well defined language with existing tool-support for analysis and verification, like timed automata. More recent papers present methods on how to use other OMG standards like the Profile for Performance, Scheduling and Time or the Object Constraint Language to obtain analyzable UML. We didn't come across preliminary research on how UML 2.0 is apt for this purpose.

In *A Real-time Profile for UML and how to adapt it to SDL*[27], Graf and Ober define a UML profile for real-time that is compatible with the OMG Profile for Performance, Scheduling and Time[12]. In contrast to the OMG profile, they put emphasis on semantics and on its use in the context of timed analysis of real-time embedded systems. The defined profile is compatible with the time concepts existing in SDL, and it is shown how the notations which do not yet exist in SDL and MSC can be adapted to those languages.

The paper *Enhancing UML-RT Concepts for Behavioral Consistent Architectural Models*[35], Engels and Küster address the issue of formal verification for UML models designed with the Rational Realtime Profile (UML-RT)[5]. They outline three drawbacks of

[5]This profile is used by Rational RoseRT

UML-RT: Inability to model behaviour of connectors, lack of semantics for the construct capsule-connector-capsule, and lack of semantics for message queues in ports. Inspired by another Architectural Description Language, Wright, it is then discussed how these issues can be resolved and how UML-RT can be made ready for formal analysis.

In *Towards UML-based formal specifications of component-based real-time software*[36], Del Bianco et. al combine UML+ and UML-RT into UML+-RT in order to provide a native construct of time and formal verification. UML+ is an extension of UML that is formally well defined and suitable for expressing the specifications of real-time systems, but does not support design and development. It's semantics are directly inspired by timed state charts.

Diethers et. al describe in *Analysis of Real-Time Systems Modeled by UML-Statecharts* how the advantages of the UML-Statecharts and Timed Automata can be combined in order to analyse and verify Real-Time systems. The UML Statechart notation is intuitive and provides concepts for hierarchical abstraction, which is missing in timed automata. On the other hand, timed automata have a clear semantics and tool support for automatic verification are available.

In *Combining UML and formal notations for modelling real-time systems*, Lavazza et. al combine UML with TRIO, a first order temporal logic, in order to make models verifiable. It is shown how subparts of the UML model can be automatically translated into TRIO, and how tool support for TRIO can be used to verify specification properties.

Another approach is chosen by Flake and Mueller. In *Specification of Real-Time Properties for UML Models*[38], the authors make use of the recently introduced Object Constraint Language (OCL) in order to model state-related time bounded constraints. The OCL was introduced by the OMG to support specification of constraints for UML models, but currently does not provide sufficient means to specify constraints over the dynamic behaviour of a model. The article presents an OCL extension that is consistent with current OCL and enables modelers to specify state related time-bounded constraints.

5.8 Conclusion

The amount of research done in the academic world and the effort shown by working groups to make UML useful to the real-time community demonstrates both the need of this community for an intuitive, yet strong and analyzable modelling language, and the inability of UML 1.x to correspond to these criteria. With the UML Profile for Schedulability, Performance and Time (SPT), a first step was made to fulfill those demands. To our knowledge, no tool-support for this profile exists at the time of this writing.

With UML 2.0, another major step is about to be made towards a UML with interesting features for the real-time developer. For example, it would be interesting to study the usefulness of the UML 2.0 timing diagram for the real-time engineer. We regret that this could beyond the scope of this thesis due to the limited time frame.

Even more interesting is going to be the combination of UML 2.0 and the SPT. If, at last, the UML Testing Profile is really going to see the daylight, the OMG might, at this stage, have come up with a complete solution to efficiently support the development of real-time applications.

However, at the time of this writing, this new and exciting developments still lie far ahead, and the UML that is supported by tools currently on the market does not meet the needs of the real-time developer, thus forcing vendors to come up with their own solutions.

Chapter 6

UML CASE Tools for RT Application Development

Various companies claim to provide UML CASE tools that are tailored for real-time development. In this section we are going to have a closer look at two of them, Rational Rose RealTime and Telelogic Tau, and try to qualify how they are apt to model our real-time application, according to the requirements that we listed in section 5.2.

6.1 Rational - Rose RealTime

Based on UML, Rational Rose RealTime (RRT) introduces additional architectural constructs, model execution and visual debugging, and complete code generation for C, C++ and Java. It is based on the Real Time Object Oriented Modelling (ROOM) language developed by Selic et. al at ObjectTime Ltd. in 1994[19]. ObjectTime Limited has later been acquired by Rational and the ROOM constructs were integrated into UML. More precisely, they were derived from more general UML concepts by using the UML extensibility mechanism. The result of this merger was presented by Selic and Rumbaugh in the influential paper "Using UML for Modeling Complex Real-Time Systems"[9] in 1998, and subsequently launched in a tool as RoseRT [1]. Its application domains are "complex, event-driven and, potentially, distributed"[9] systems.

[1]In literature, the above paper is sometimes referred to as UML-RT or the UML Real-Time profile. This may cause confusion since the OMG UML Profile for Schedulability, Performance and Time (2002) is also referred to as UML-RT in literature. IBM Rational white papers refer to the paper as the RRT profile, which stands for the Rational RealTime profile. This profile is not OMG adopted; its concepts rather inspired the design for UML 2.0.

6.1.1 The Modelling Language

In this section discuss the real-time relevant extensions to UML. The most obvious extension are the *Capsules, Ports, Protocols* and *Connectors*, as well as a novel diagram, the *Capsule Structure Diagram.* All of these concepts are semantically very similar to their relatives in UML 2.0.

A capsule is an active object that communicates with its environment exclusively via messages through its ports. Valid messages are defined by a protocol that is associated with a port. Two communicating ports are linked via a connector which models a signal based communication channel.

By forcing capsules to communicate solely through ports, it is possible to fully de-couple their internal implementations from any direct knowledge about the environment. This makes them highly reusable.

Protocols and Connectors

A protocol is a specification of desired behaviour that can take place over a connector. It specifies valid incoming and outgoing signals, and comprises a set of participants, each of which plays a specific role in the protocol. A *binary protocol*, involving just two participants, is the most common protocol and the most simple to specify. It defines two roles, the *base role* and the *conjugate role.* The message set for the conjugate role can be simply obtained by inverting the incoming and outgoing signals from the base role. These roles are therefore said to be *complementary.*

Connectors model a communication channel for transmitting a particular protocol. They can only interconnect ports that play complementary roles in the associated protocol.

Ports

Ports are objects whose purpose is to act as boundary objects for a capsule instance, and is the only means for a capsule to communicate with its environment. They are owned by the capsule instance in the sense that they are created along with their capsule and destroyed when the capsule is destroyed.

Each port is associated with a specific protocol role.

There are two type of ports: *relay ports* and *end ports.* Relay ports simply pass all signals through to sub-capsules, and therefore provide a mean for sub-capsules to communicate with the outside world without being exposed to it. End ports are connected to the capsules state machine. They are the ultimate sources and sinks of all signals sent by capsules, and contain a queue for messages that have been received but not yet processed by the state machine. End ports can be either *public* or *protected.* Public end-ports are

	Classes	Capsules
Communication	Public operations	Messages, received through public *ports*, defined by a *protocol*. Messages are the only means for a capsule to communicate with its environment.
Attributes	Public, private, protected	Only private, to enforce encapsulation
Behaviour	Method implementation	Defined by state machines which run in response to the arrival of messages

Table 13: Specialized properties of capsules in comparison to classes

boundary objects, whereas protected end-ports are used by the capsule's state-machine to communicate with sub-capsules or runtime services, e.g. an operating system service.

Capsules

Capsules are the fundamental modelling constructs of RRT. They are complex, physical, possibly distributed architectural objects that interact with their surroundings exclusively through one or more ports. As a stereotype of the UML concept "class", capsules have much of the same properties as classes. For example, they have operations and attributes, and may participate in dependency, association and generalization relationships. However, they also have several specialized properties that distinguishes them from classes. These properties are given in table 13.

Capsules have their own thread of control; they can be seen as highly encapsulated active objects. A capsule may contain sub-capsules, joined together by connectors. This sub-capsules cannot exist independently of the capsule.

The behaviour of a capsule is defined by its state-machine. This state-machine can send and receive signals via end-ports of the capsule, and is the *only* entity that can access the internal protected parts in its capsule. A system built with capsules can therefore be seen as a network of interoperating state machines. State transitions are triggered by the reception of a signal and have *run-to-completion* semantics. This means that an action initiated by the arrival of an event (e.g. to a state transition) is not interrupted, even if an event of higher priority arrives. Therefore, modification by the high priority event of internal variables that were in the process of being modified can be prevented.

A capsule can have a specific *role*, which is basically a name given to a capsule participating in the context of another capsule's specification. The behaviour of a capsule is modelled in the *state transition diagram*, while its (hierarchical) structure is modelled in the *capsule structure diagram*. Both diagrams have a clearly defined semantics, and are

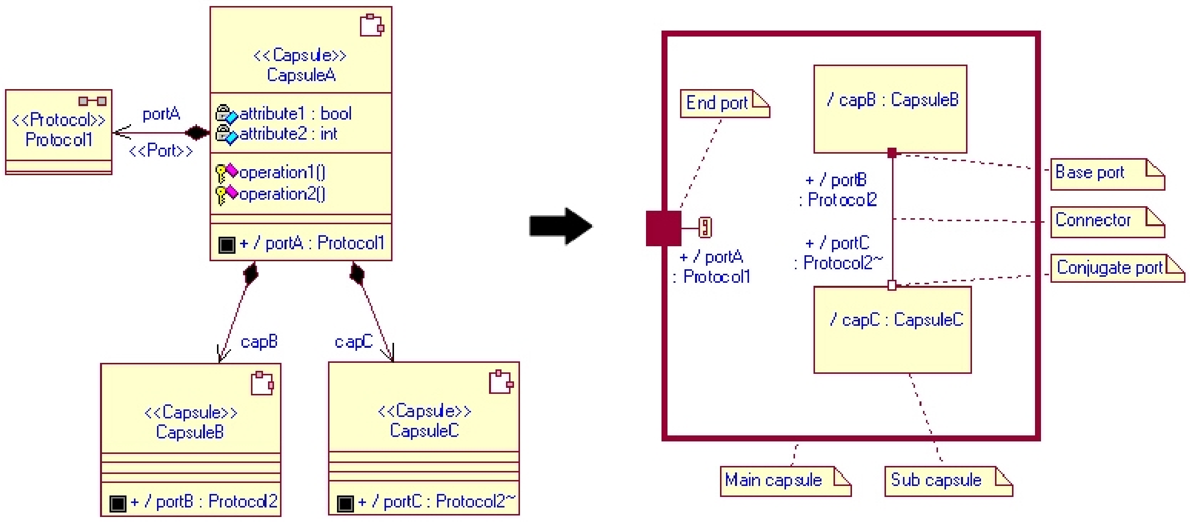


Figure 9: Simple class diagram (left) and corresponding capsule structure diagram (right).

important for code generation.

Capsule Structure Diagram

A capsule structure diagram is a specialized collaboration diagram. It shows the capsule roles, the ports and how they are related via connectors. It also shows the capsule boundary, which is a visual representation of the capsules encapsulation shell. It describes the *architecture* of a network of capsules. Communication relationships between capsule ports are explicitly shown. In contrast, in a general collaboration, communication between objects is modeled using an association role between two classifier roles. From this diagram, structural code will be generated.

An example is given in Fig.9, which shows a simple class diagram (left) and corresponding capsule structure diagram (right). CapsuleA (A) aggregates 2 sub-capsules, CapsuleB (B) and CapsuleC (C). In their interaction with A, B and C have the role names capB and capC respectively. A has an end port, portA. This port implements the base-role for Protocol1. portB and portC belong to B and C respectively. portB has the base-role and portC the conjugate role in the interaction specified by Protocol2 (which is not shown in this class diagram).

Time

RRT comes with a run-time library for real-time services, including services for controlling concurrency execution of finite state machines, for delivering messages, and for timing and logging. The timing service converts time into events and can be accessed from within a capsule through a protected port. It provides one-shot and periodic timers.

Connectors also incorporate the notion of time, by allowing the simulation of a fixed transmission delay.

6.1.2 Code Generation

Code generation is based on the information provided by the capsule structure diagram and the statechart diagram. An important point for code generation is the definition of an *action language*. With an action language, one can specify actions and associate them to certain parts of the diagram, e.g. to state transitions, state entries, object creation etc. In RRT, the action language is the target programming language, i.e. C, C++ or Java. Thus, actions are directly described in the programming language for which code is going to be generated.

By default, all capsules are mapped onto a single operating system thread, and parallel behaviour is enabled and controlled by the run-time library of RRT. However, RRT is designed to easily allow mapping of capsules onto several OS threads.

6.1.3 Supported Diagrams

The supported diagrams are: Use case diagram, Class diagram, State diagram, Collaboration diagram, Capsule structure diagram, Sequence diagram, Component diagram and Deployment diagram.

6.1.4 Conclusions

In the following we describe how RoseRT meets our requirements for a CASE tool.

1. *Visualization and modelling of time and time constraints.* The only means to visualize time and time constraints is through annotation of behavioural diagrams. This is useful to visualize local constraints, but insufficient in order to understand a network of constraints, and how they might influence each other.
2. *Modelling of concurrency and distribution.* The capsule construct allows modelling of concurrency in a very elegant way. Distribution can be modeled in the deployment diagram.

The Connexis feature of Rational Rose supports the modelling of distributed, communicating components. With Connexis, one can establish communication paths and send messages between capsules in separate processes whether they reside on the same node or on separate nodes. Connexis uses the TCP/IP stack of the underlying operating system to achieve this.

3. *Testing and model-verification.* RRT supports model execution. While executing it, the behaviour of the system can be observed directly in its model view. The model can therefore be debugged visually, i.e. by watching how the system moves from state to state and observing the values of internal variables. Through model execution, the designer can obtain an early feedback of the modeled behaviour. He also has something to "play" with, which makes developing more interactive.
   Message flows through ports can be traced, and messages can be injected into a capsule by hand. In general, we found that graphical debugging is a very powerful way of debugging software; a graphical model is much more expressive than source files.
   A drawback is that RRT does not allow testing and verification of time-focused constraints. Further, it is not possible to perform any formal analysis like protocol deadlock or timing analysis. Note that Rational proposes Quality Architect for extended testing support. We did not use Quality Architect because we didn't have this product.
4. *Code generation.* Code generation is possible. Generated code is fairly well readable. One big advantage of code generation is that the model maps one to one to the code. Rational suggests that the whole behaviour of the application is directly to specify in the model. In their philosophy, the designer shouldn't touch the source code of the application. The support for this method of designing a system, called Model Driven Architecture(MDA), could be better resolved. While simple behaviour can be implemented very easily, complex behaviour, like integration with model-external parts (like the Windows Sound API in our case) seemed fairly complicated to us.

Rational RoseRT claims to be suitable for real-time development, which is an exaggerated statement. It supports very well the development of embedded systems (which is often found in the real-time world) by providing architectural constructs that are specifically designed to support encapsulation.

RoseRT focuses very strongly on capsules. This might not always be perceived as positive by the designer, as it restricts the ways an application is modelled.

By modelling our problem domain, we could clarify issues how the different components, the recorder, sender, receiver and player can be controlled and how they communicate. Even more important, it helped us to understand the functionality of certain complicated parts of the system (e.g. the player, as we will see later), even though they were only modelled on a high level: we did not integrate the sound device API.

In general, we thought that the "touch & feel" of RRT is good, and we liked modeling with this tool.

6.2 Telelogic - Tau Generation 2

Telelogic Tau Generation 2 Developer (TauG2), version 2.1, is the latest product in the Telelogic CASE tools family. Like Rational Rose Realtime, it can execute models, verify the application and generate code.

To achieve this goals, TauG2 merges UML and concepts from the emerging UML 2.0 standard with the Specification and Description Language (SDL). According to the tool documentation, Tau uses the extension mechanism of UML to add a number of extensions to UML. Some of these extensions are Tau specific and some are defined by ITU (International Telecommunication Union) Recommendation Z.109 "SDL Combined with UML".

The most obvious evidence of UML 2.0 integration are: ports, connectors, active classes with component character (ports only mean for communication), connectors and a new diagram that maps to the composite structure diagram of UML 2.0. The most obvious evidence of SDL integration is the replacement of the UML statechart by the SDL state machine.

In the past, Telelogic used to propose two different tools for UML and SDL development. High-level analysis and design could be done in UML, and was then translated into SDL, where the model could be refined and tested. In order for the translation mechanism to produce correct SDL, the UML models had to be designed according to a certain semantics. This was negatively perceived by designers [1], as translation from UML to SDL was error prone and needed to be corrected manually. With the new tool, this is elegantly circumnavigated by directly designing SDL state machines.

SDL and the ITU-T Recommendation Z.109

SDL [26] is a formal language for specifying reactive systems. It is event-driven, and was mainly used in telecommunications development. The language is semantically defined and comes with a complete action language, so that code generation is possible.

SDL is a formal, object oriented and highly testable language. It is primarily intended to be used to specify complex, distributed applications. Such complex applications would involve many concurrent processes that communicate using discrete signals. SDL, being a formal language, provides an unambiguous system specification. In addition it is hierarchical in nature, allowing information hiding and abstraction. An SDL specification

can be graphical (SDL/GR) or in a textual phrase representation (SDL/PR). SDL/PR can be seen as being a high level programming language.
SDL increasingly used for development of real-time and embedded systems, where the functional behaviour is time dependent.

The ITU-T recommendation Z.109 defines a specialisation of a subset of UML that has a one-to-one mapping to a subset of SDL. The semantics of this specialisation is given by the semantics of the corresponding SDL. The intention behind this recommendation is not to use the specialised UML *instead* of SDL, but to use SDL *combined* with UML, to take advantage of the strengths of the two approaches[24]. For example, UML can be used for use case and collaboration modelling, and than turn to SDL when precise semantics, for example in specifications of actions, is needed.

This ITU Recommendation is currently being updated to be consistent with UML 2. Finding a mapping between UML 2.0 and SDL is going to be more easy; the revisions in UML 2.0 to classes with internal structure and interactions have been semantically aligned with ITU-T SDL and Message Sequence Charts(MSC)[11].

6.2.1 The Modelling Language

In this section we present relevant features of the modelling language of TauG2. The constructs that are found are very similar to or the same as their equivalents in UML 2.0.

Connectors

A connector specifies a medium that enables communication between parts of an active class or between the environment of an active class and one of its parts. They are most of all represented for visualization. If omitted, necessary connectors will implicitly be created between ports. If a port has explicit (visualized) connectors, no implicit connectors will be connected to the port. A connector may be uni-directional or bi-directional and specifies for each direction the allowed information.

A certain delay can be specified on a connector.

Interfaces

An interface groups a set of attributes, operations and signals which must be implemented by the class that implements the interface. Interfaces can also be associated to each other to provide a definition of *protocols* (or contracts between classes) that realize the involved interfaces.

Ports

Ports are the interaction points of an active class with its environment. They specify the implemented interface (realized) and the needed interfaces from other classes (required). They are only allowed on active classes.

Ports come in two different kinds: *behaviour* ports and *non-behaviour* ports. A behaviour port is directly associated with the state machine of the class, whereas a non-behaviour port relays the outside with internal parts of the active class (the active subclasses).

Active Classes

The active class is the fundamental building block for modeling real-time behaviour with TauG2. It has its own flow of control and can both initiate behaviour and passively react to behaviour as observed on its interfaces. Active classes define the structure (architecture) and behaviour of a model, and they communicate with their environment through *ports* that realize or require certain interfaces. All instances of active classes execute asynchronously and concurrently with the other parts.

The *structure* on an active class is defined in one or several *architecture diagrams.*

The *behaviour* of an active class is defined by a State machine defined by one or several Statechart diagrams.

An active class may have attributes and operations. Public attributes and operations are those who are declared by an interface that is realized by an active class.

Architecture Diagram

An architecture diagram defines the internal run-time structure of an active class, in terms of other active classes. These building blocks are referred to as parts since they always must be composite parts of the containing class. Furthermore the parts are also restricted to be instantiations of active classes. Architecture diagrams may also express the communication within the active class by visualizing connectors between the communication ports of the parts. We can thus describe complex architectures by splitting it hierarchically.

State Machine

The TauG2 tools don't use UML state charts, but SDL state machines. Even though there exists a mapping between UML and SDL state machines at the meta-model level, the graphical representation is very different. UML state charts show how an object moves from one state to another and the rules that govern the change of state. Their focus is to give a good *state* overview. SDL state machines, on the other hand, focus on transitions and their internals, i.e. the actions accomplished and the control of flow during a transition.

State machines can cooperate by sending asynchronous signals to one another. State transitions happen as a result of external stimuli. A state machine is the only entity that can access and modify data that is local to the component that embeds the state machine.

Time

TauG2 accesses time through the *SDL timer* object, which are so called *one shot* timers. They cannot be periodic, and they express clocks which "tick" periodically, where ticks are consumed intstantanously or lost [27].

6.2.2 Supported Diagrams

The supported diagrams are: Class diagram, Architecture diagram, Statechart diagram, Sequence diagram, Use Case diagram, Text diagram (to specify code).

6.2.3 Differences between UML 2.0/Z.109 and TauG2

The tool documentation states that since neither UML 2, nor Z.109, were finalized at the time of designing Tau 2.1, some differences between the implementation and the current proposals for UML 2/Z.109 exist. This includes:

- The *architecture diagrams* of Tau 2.1 are in UML 2 called composite structure diagrams. They are not (yet) part of Z.109 either.
- The *use case diagrams* of Tau 2.1 represent that status of the UML 2 proposal from summer 2002 and have not been updated to reflect later changes. The difference is mainly that Tau 2.1 use case diagrams allow several occurrences of each actor and the relationship between an actor and a use case is called performance. In UML 2 the diagram supporting multiple occurrences of each actor is called use case collaboration diagram and the relationship is a connector.
- The *import relation* is in Tau defined by the *use* dependency. This was removed from the UML 2 submission late 2002, but is still part of Tau.

Our own assessment is that TauG2 adopted exclusively the architecture related concepts from UML 2.0, and none of the new features related to the activity diagram, the sequence diagram, the timing concepts, etc.

6.2.4 Conclusions

In the following we describe how TauG2 meets our requirements for a CASE tool.

1. *Visualization and modelling of time and time constraints.* Time can be modeled through SDL-timers. The fact that SDL timers cannot be periodic limited their usefulness for our application (where modelling of periodically generated/consumed sound data is necessary). Time constraints are not inherent to modelling language of TauG2; the only means to visualize them is through model annotation.
2. *Modelling of concurrency and distribution.* Concurrency and distribution is well supported by the UML 2.0 Component construct. For the high level design of concurrent processes, we particularly missed the UML Activity diagram, which allows easy visualization of the logic of concurrent activities.
3. *Testing and model-verification.* TauG2 allows model execution and visual debugging. This has the same powerful advantages as already mentioned for RoseRT. In addition, TauG2 comes with Tau/Tester, a test suite based on the Test and Test Control Notation 3 (TTCN-3). TTCN-3 allows the creation of (textually described) test cases which are graphically represented as Message Sequence Charts (MSC). TTCN-3 has been successfully used in numerous fields, but it has had its main successes in testing (tele)communications protocols. Its focus is on testing functional requirements.

   For the handling of time, TTCN-3 provides a timer mechanism. This timer mechanism is not sufficient for real-time testing, as stated by Dai, Grabowski and Neukirchen in [32]:

   "The timer mechanism is designed for supervising the functional behaviour of an implementation under test, e.g., to prevent the blocking of a test case or to provoke exceptional behaviour. But, it is too slow and too clumsy for the test and measurement of real-time properties, because the measurement of durations is influenced by the TTCN-3 snapshot semantics and by the order in which the port queues and the timeout list are examined. TTCN-3 makes no assumptions about the duration for taking and evaluating a snapshot. Thus, exact times can not be measured. Furthermore, TTCN-3 has no concept of absolute time, i.e., a test component cannot read and use its local system time. In real-time testing, the absolute time is necessary to check relationships between observed test events and to coordinate test activities."

   They consequently suggest *Timed*TTCN, which introduces the missing concepts. TauG2 implements none of them.
4. *Code generation.* TauG2 states that it allows C and C++ code generation. It seems important to note that C++ code generation is only supported for passive objects, and thus the above statement is not entirely true. Active objects are always first

translated to SDL, from where C code generation is generated. The generated code was hard to read.

As with RoseRT, TauG2 does not meet our requirements for specifying and verifying hard real-time constraints. System testing is more developed in Tau/Tester with TTCN-3 compared to RoseRT which ships by default with very poor testing support.

The integration of SDL state machines into the tool was positively perceived, as it allows to keep a better overview on what is going on within a state-transition. SDL state machines can be split up, which allows them to be visualized in small parts. This scales better with state machines that capture complex behaviour.

In general, the UML 2.0 alignment of TauG2 has been perceived as positive. As far as we could see, mostly concepts used to enable component based development have been borrowed from the emerging standard, which was a bit of a deception. Compared to RoseRT with its strong capsule focus, TauG2 leaves the designer more freedom on what kind of system he wants to develop, which we perceived as positive.

It is not possible to model and verify hard real-time constraints with TauG2.

6.3 Discussion and Conclusion

The modelling language proposed by TauG2 and RoseRT are very similar. This is not surprising, as TauG2 is based on (an early proposition for) UML 2.0, for which the Rational RealTime Profile - implemented by RoseRT - is one of the foundations.

The support for timing analysis is weak in both tools, and our requirements in this regard have not been met. Our expectations were high: In early implementation versions we experienced difficulties in scheduling communicating threads in a time-efficient and correct way. By use of a real-time tool, we indeed expected some analyzing-support to understand the nature of the problem. We realized that to support this, a tool must not only be able to model and *simulate* the environment in which a application is run (for example, how many processes run parallel to the process in consideration), but also low level things such as thread scheduling policies and system performance. Tools that support this kind of analysis were only found in the academic world, which might give an intuition of their complexity and difficulty to handle.
With the Profile for Schedulability, Performance and Time, an support for the development of such tools has recently been established. It is going to be interesting to see how far this profile is going to be accepted and integrated by tool-vendors.

The lack of support for analysis of real-time issues was a major reason why the voice component was not completely code-generated with one of the tools. We deal with very small time-spans, hence there is a high risk that some scheduling deadline is not met.

As we could not assure that the modelled system was meeting these constraints, it was highly probable that an automatically generated application would have needed to be debugged "by hand". Implicit and non documented semantics of behaviour, like the handling of the message queue in a port, and generated code that is difficult to read, especially in the case of TauG2, made us suspicious that debugging would exceed the time frame allocated for modelling and implementation. Thus, no code was generated.

This leads directly to another point of critique, which is the support to model concurrent behaviour if the Component (or Capsule) construct is not quite the construct that is needed by the designer. After having decided not to generate code from the model, we wanted to model the voice component according our own analysis of what requirements must be met by the model to obtain a sound overall behaviour. We wanted to be able to model multiple, sometimes communicating threads, that do *not* necessarily run in an completely encapsulated environment. The UML Activity Diagram is quite helpful in this regard. Luckily it is supported by RoseRT. It is not supported by TauG2. But still, a lot of things where found to be missing in the modelling language of both tools. It must be said that most of them are due to the lack in the specification of UML.

The OMG Final Adopted Specification of the UML 2 Superstructure [10] comes with all of the features that we identified to be missing for the design of the voice component. It's advanced abilities to structure sequence diagrams (e.g. model loops, alternatives etc.), time (e.g. the concept of time constraint that is native to UML 2.0), and the new semantics for actions and the activity diagram (extended support to model concurrency, support to define a set of actions that can be interrupted by an event, model exceptions etc.) would have helped us considerably in the design phase. It is a pity that TauG2 only adapted the component related concepts from UML 2.0. If UML 2.0 based CASE tools are going adopt the OMG Specification, they are going to have very interesting and exciting new features. We hope to see these tools on the market soon.

Chapter 7

Design

In this chapter we present two models for the ChattaBox voice component. The first model follows a component based design approach using capsules. The second model is standard UML, and we annotate the UML activity diagram to express time constraints.

7.1 Design with Capsules

In the first iteration, we tried to model the voice component following the RoseRT design philosophy, i.e. using capsules and architectural diagrams. The goal was to obtain a realistic executable model to make an assessment of RoseRT's analyzing capabilities. The diagrams used were the class diagram, capsule diagram and state diagram. We present selected parts of the model.

7.1.1 Use Case Diagram

The use case diagram is shown in Fig.10. The actors are the entities that directly interact with the voice component. The CommunicationManager can be seen as the application using the voice component. The SoundDevice provides information to the component when and audio data block has finished recording or playing. The NetworkDevice informs whether new data was received or data was successfully sent.

7.1.2 Class Design

The class design is shown in Fig.11. The design of the VoiceModule capsule is inspired by the Controller Pattern described in [25]. The control part is separated from the functional part. The capsule aggregates a SystemController capsule and four functional component capsules: the SoundRecorder, the RTPSender, the RTPReceiver, and the

SoundPlayer. These components extend an AbstractFuntionalComponent (Fig.13).The AbstractFuntionalComponent defines a basic structure and a state machine that can be controlled via the ControlProtocol. This protocol defines the messages *configure, start* and *stop*, and the order in which they must be sent. The SystemController and the SoundPlayer aggregate a timer in the form of a protocol and a port. The SystemController uses a periodic timer for synchronization with its peer controller at the other side of communication. The SoundPlayer uses it to know when to schedule sound-data for playback.

7.1.3 Architecture of the Voice Component

In Fig.12 we demonstrate how the capsules described in the class diagram are architectured within the VoiceComponent Capsule. The SystemController obtains messages defined by the ControlProtocol through the applicationInterface. It uses the same protocol to communicate with the functional components through the controlInterface. As soon as both communicating system are operational, the SoundRecorder will obtain sound through its soundDevice port and relay it to the Sender which will send it to the peer over the network. The RTPReceiver will pick up data from the network and relay it to the SoundPlayer, which will, when scheduled, relay the data to the sound device through the toSoundDevice port.

The VoiceComponent capsule provides logic for streaming voice over a network. It does not implement the wrapper for the sound device and the network device, since the implementation of these wrappers change depending on the underlying operating-system. The services that the VoiceComponent requests from its environment is shown by its public ports and their associated protocols. In our case, we could imagine three other capsules, which, combined with the VoiceComponent, provide all services needed by a complete VoIP application. The first capsule either wraps or contains the application which presents interfaces to the user. A second capsule wraps the sound-driver of the operating system and provides its services to the capsule environment. A third capsule does the same for the networking part of the operating system.

7.1.4 Architecture and Behaviour of the SoundPlayer Capsule

The SoundPlayer capsule illustrated in Fig.14 is responsible for providing a stream of voice data to the sound device, i.e. the sound device capsule. It inherits from the AbstractFunctionalComponent a state machine with three states - Unconfigured, Configured and Operational - and a structure containing one port. This port is for communication with the SystemController, who configures, starts and stops every functional component. The Operational state of the state machine is extended with a behaviour that satisfies its functional requirements. The structure is extended with three ports, one for communication with the RTPReceiver, one for communication with the sound device, and one internal port that accesses a timing service. This is shown in Fig.14 and 15.

As soon as it is operational, the SoundPlayer capsule waits for packets to arrive. With the first packets it fills up a jitter buffer. As soon as the jitter buffer is full, it enters the play state and starts to play data. From now on, sound-data is periodically sent to the sound-device, no matter if there is enough data in the jitter buffer or not. This is guaranteed through the playPacket and playLastPacket state transistions, which are timed. If there is no data in the jitter buffer, the capsule goes into the recoveryFromInterruption state and continues to play soundData (the last succesfully played packet). It exits from this state only when a packet is received whose playback time lies in the future.

7.1.5 Discussion

The goal with this design iteration was to obtain a realistic model that contains time-critical behaviour and is executable. It is not complete, because the capsule based design approach was abandoned due to reasons mentioned in section 6.3. We think the model illustrates a capsule based design approach well, and might give the reader an idea of how modelling with RoseRT can look like.

When executing the above model, transitions between states can be observed directly in the state diagram. Message transitions through ports can be traced and values of capsule internal variables can be displayed. Messages can be manually injected in ports. This is advantageous for debugging the system, as a capsule can be isolated and their execution be studied apart from the rest of the system.

The state diagrams are state oriented, compared to SDL state charts that are transition oriented. This means, that we have a good overview of the states, but cannot visualize the actions that are executed within a transition. It would be nice to be able to visualize these actions.

Finally, even though this approach was abandoned, we consider the time that was spent as a valuable investment. The component based thinking pattern helped us to structure tasks and responsibilities. Even though the design approach in the final model is quite different, it was inspired by the ideas developed at this iteration.

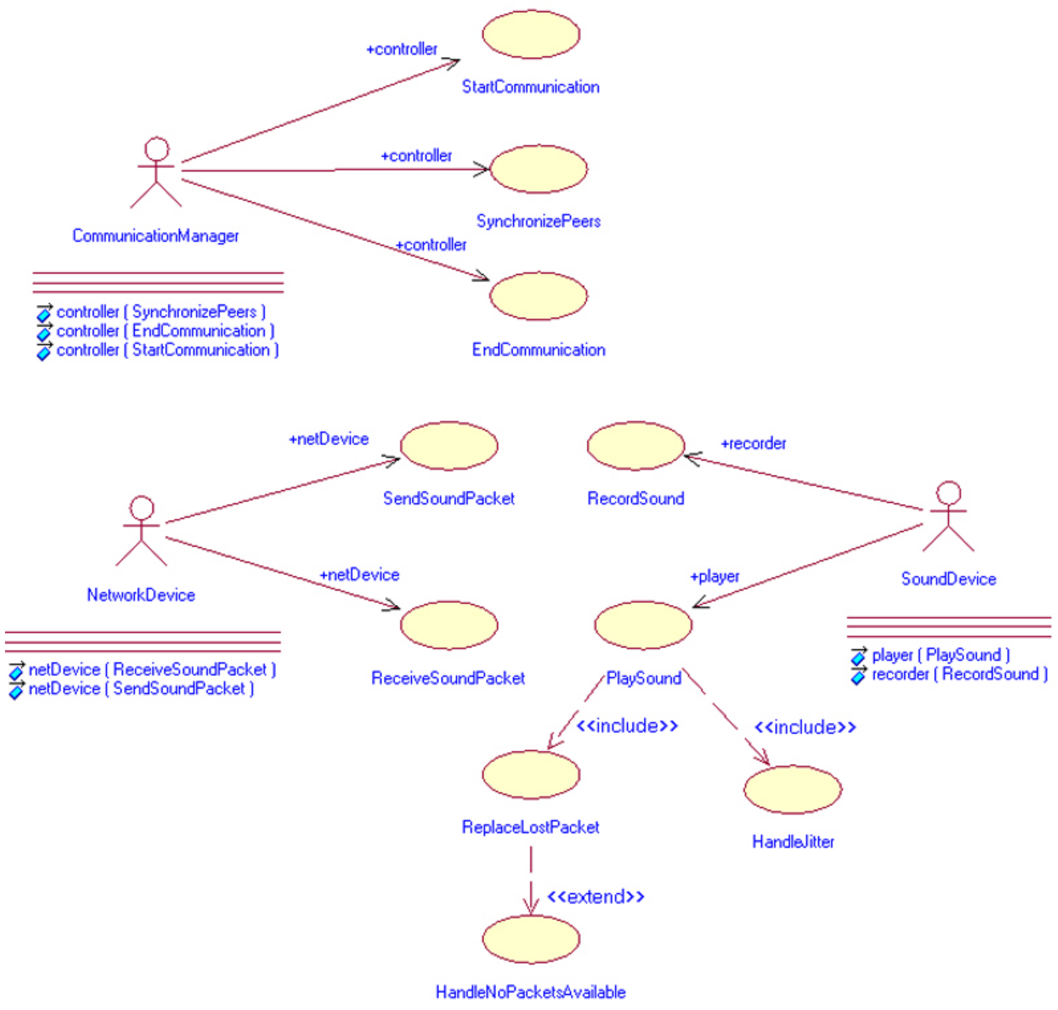
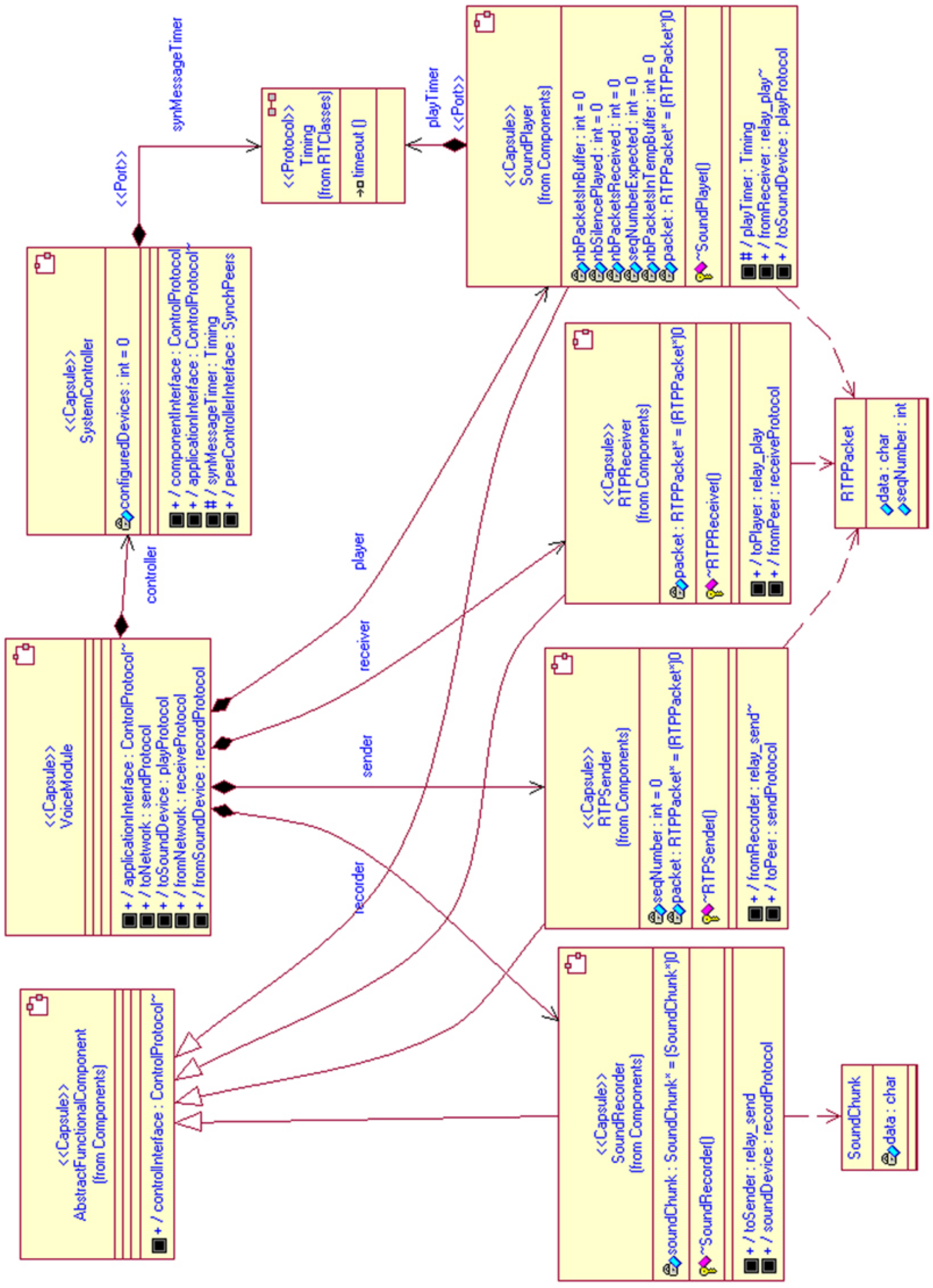


Figure 10: The use case diagram for the capsule design.



<<Capsule>> SystemController
configuredDevices : int = 0
+ / componentInterface : ControlProtocol
+ / applicationInterface : ControlProtocol~
# / synMessageTimer : Timing
+ / peerControllerInterface : SynchPeers
<<Port>>
synMessageTimer
<<Protocol>> Timing (from RTClasses)
timeout ()
playTimer
<<Port>>
<<Capsule>> SoundPlayer (from Components)
nbPacketsInBuffer : int = 0
nbSilencePlayed : int = 0
nbPacketsReceived : int = 0
seqNumberExpected : int = 0
nbPacketsInTempBuffer : int = 0
packet : RTPPacket* = (RTPPacket*)0
~SoundPlayer()
# / playTimer : Timing
+ / fromReceiver : relay_play~
+ / toSoundDevice : playProtocol
controller
<<Capsule>> VoiceModule
+ / applicationInterface : ControlProtocol~
+ / toNetwork : sendProtocol
+ / toSoundDevice : playProtocol
+ / fromNetwork : receiveProtocol
+ / fromSoundDevice : recordProtocol
player
receiver
sender
recorder
<<Capsule>> RTPReceiver (from Components)
packet : RTPPacket* = (RTPPacket*)0
~RTPReceiver()
+ / toPlayer : relay_play
+ / fromPeer : receiveProtocol
RTPPacket
data : char
seqNumber : int
<<Capsule>> RTPSender (from Components)
seqNumber : int = 0
packet : RTPPacket* = (RTPPacket*)0
~RTPSender()
+ / fromRecorder : relay_send~
+ / toPeer : sendProtocol
<<Capsule>> AbstractFunctionalComponent (from Components)
+ / controlInterface : ControlProtocol~
<<Capsule>> SoundRecorder (from Components)
soundChunk : SoundChunk* = (SoundChunk*)0
~SoundRecorder()
+ / toSender : relay_send
+ / soundDevice : recordProtocol
SoundChunk
data : char

Figure 11: Class Diagram for the capsule design.

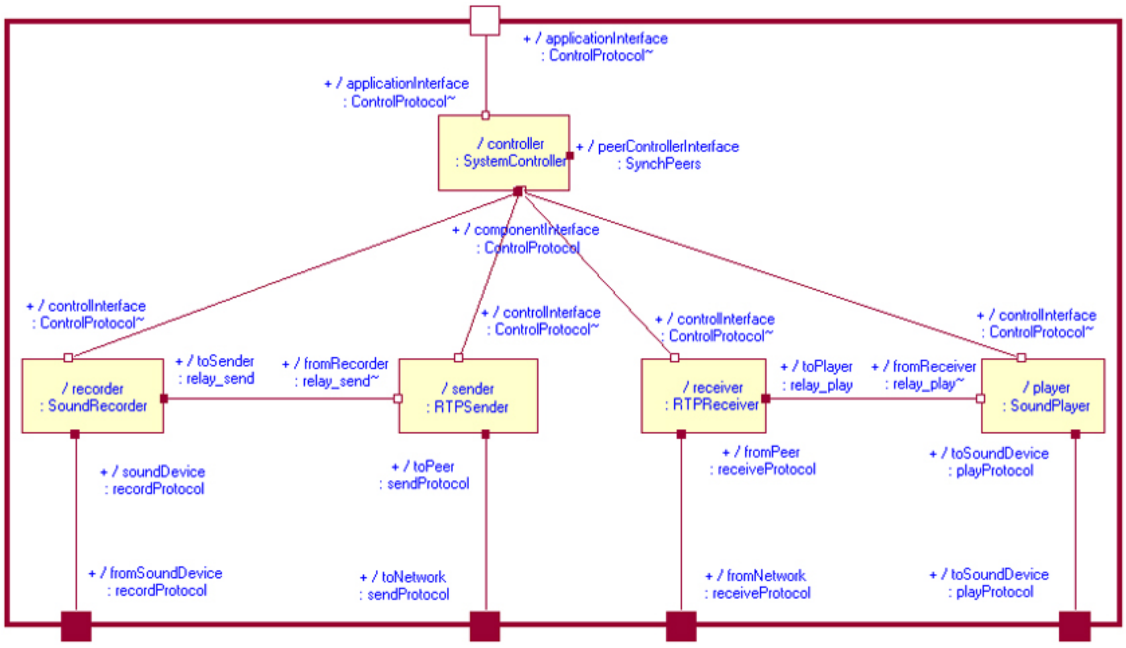

+ / applicationInterface : ControlProtocol~
+ / applicationInterface : ControlProtocol~
/ controller : SystemController
+ / peerControllerInterface : SynchPeers
+ / componentInterface : ControlProtocol
+ / controlInterface : ControlProtocol~
+ / controlInterface : ControlProtocol~
+ / controlInterface : ControlProtocol~
+ / controlInterface : ControlProtocol~
/ recorder : SoundRecorder
+ / toSender : relay_send
+ / fromRecorder : relay_send~
/ sender : RTPSender
/ receiver : RTPReceiver
+ / toPlayer : relay_play
+ / fromReceiver : relay_play~
/ player : SoundPlayer
+ / soundDevice : recordProtocol
+ / toPeer : sendProtocol
+ / fromPeer : receiveProtocol
+ / toSoundDevice : playProtocol
+ / fromSoundDevice : recordProtocol
+ / toNetwork : sendProtocol
+ / fromNetwork : receiveProtocol
+ / toSoundDevice : playProtocol

Figure 12: The structure of the VoiceComponent capsule.

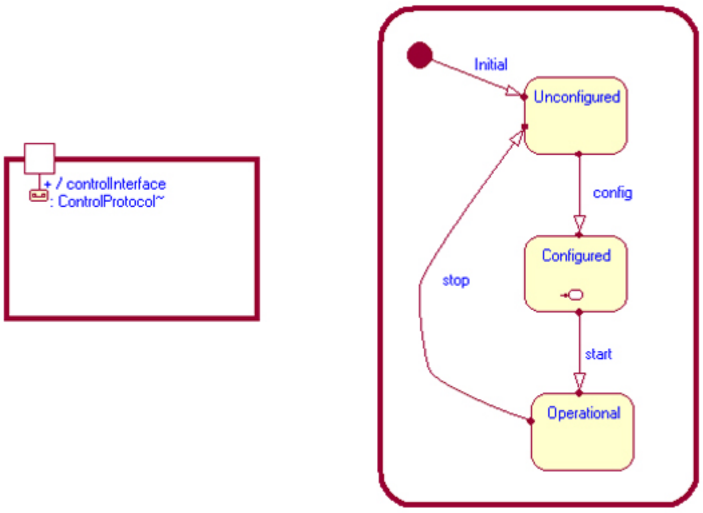


Figure 13: Structure and state machine of the AbstractComponent capsule.

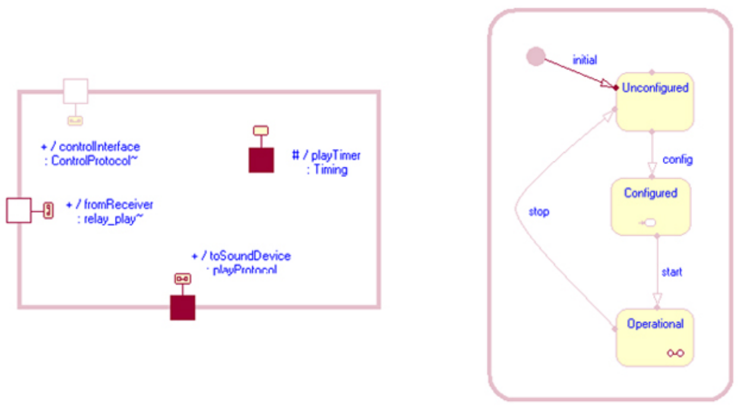
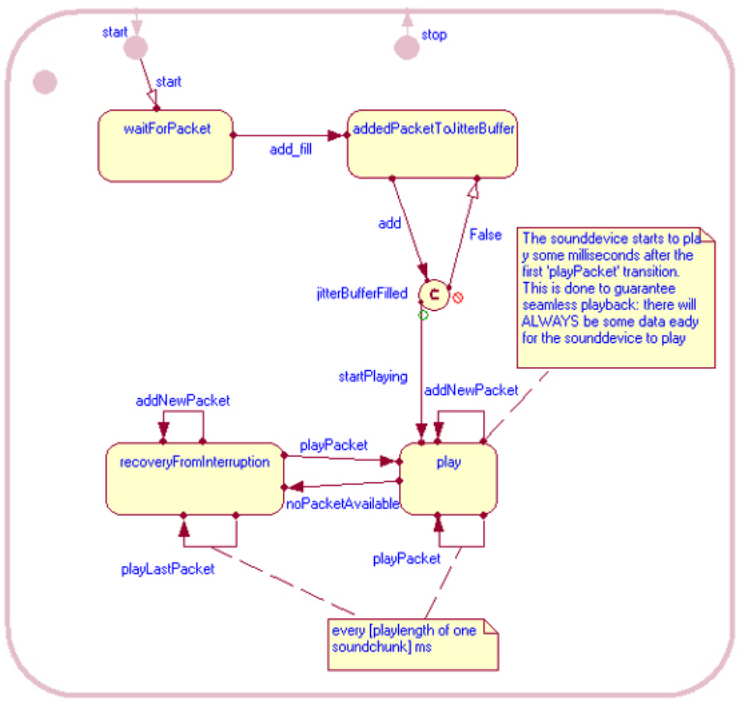


Figure 14: Structure and state machine of the SoundPlayer capsule.



The sounddevice starts to pla y some milliseconds after the first 'playPacket' transition. This is done to guarantee seamless playback: there will ALWAYS be some data eady for the sounddevice to play

every [playlength of one soundchunk] ms

Figure 15: The internals of the Operational state of the SoundPlayer capsule.

7.2 Final Design

The final design is significantly different from the previous design. Capsules are not used. Behaviour is specified in the form of sequence diagrams and activity graphs, compared to state charts in the previous design. The goal for this iteration was to clarify how the final implementation should look like, and to create documentation artifacts. Model execution was not aimed. Code generation was only used to obtain skeletons for the C++ classes of the package.

7.2.1 Real-Time Constraints

We chose the activity diagram to illustrate our real-time constraints. The annotated diagram is shown in Fig.19.

In order to identify our constraints we introduce the UML-stereotype `<<RT constraint>>`. If this stereotype is followed by the keyword `#ref`, there exists an model external description for the constraint. The constraint is written in a UML-textbox and associated with the activities and/or activity states that involved.
We further adopt the two behavioural UML 2.0 meta-classes `TimeConstraint` and `TimeObservationAction`. A `TimeObservationAction` is an action that, when executed, returns the current value of time in the context in which it is executing.

Every Activity has two `TimeObservationAction`: `Activity.enter` and `Activity.leave`. They describe the instant when an Activity is entered or left respectively. ActivityStates are Activities that include a waiting state. They are described by the same symbol as a state in a statechart diagram. Every Activity has four `TimeObservationAction`. `ActivityState.enter` and `ActivityState.leave` describe the instants when an ActivityStates is entered or left, and `ActivityState.wait.enter` and `ActivityState.wait.leave` describe the instants when the waiting state of an ActivityState is entered or left.
The time constraint between two `TimeObservationAction` is described by a `TimeConstraint`. The key-words `hard` and `soft` following a `TimeConstraint` indicate the time-margin for the constraint.
Beside that, we define `ThreadInvokation` which is an attribute of a swim-lane in an activity diagram. It defines the maximum time duration between two invocations of the thread executing the activities in the swim-lane. At each invocation, the thread executes at least one action.

All the three threads have an activity state. The waiting states correspond to recorderWait, receiverWait and playoutWait respectively.

The constraints with a reference number are detailed in Tab.14. The constraints on thread invocation are given directly in the diagram.

7.2.2 Structural Diagrams

Class Diagram

The class diagram (Fig.16) shows the eleven classes that make up the voice component package.

Class VoiceComponentCtrl This class controls the whole voice component. It is the only active class in the package. The application using the voice component starts and stops streaming voice by calling the `startSession()` and `endSession()` methods.

When the class receives the `startSession()` call, it creates three sub-threads. The first controls the recording and sending behaviour with the `recordSendCtrl()` method, the second controls the receiver behaviour with the `receiveCtrl()` method, and the third controls the playout behaviour with the `playCtrl()` method. The main thread waits until it receives the stop signal from the application layer. It then stops the three threads and exits the voice component. This behaviour is illustrated in the activity diagram in Fig.19.

Class SoundRecorder The SoundRecorder class collects ADBs from the sound device. Its main method, `getToAudioBuffer()`, provides automatic voice activity detection on the ADBs and inserts them into a buffer queue, the AudioBuffer, from where they can be accessed by other parts of the application.

Class RtpSender The RtpSender class makes sure that the audio data is safely sent over the network. It wraps an ADB into an RTP Packet, fills the RTP Header with the necessary information and provides a mechanism for compression and Forward Error Correction. It's main method, the `sendFromBuffer()`, does this by collecting ADBs from an AudioBuffer. This method can be configured to enter an efficient wait state if the AudioBuffer is currently empty. It is then notified by the AudioBuffer as soon as new data is available.

Class RtpReceiver The RtpReciver class collects RTP packets from the network, decodes them and applies FEC if necessary. The method `receiveToJitterBuffer()` does this by collecting one packet at a time, and putting the data into a JitterBuffer.

Class SoundPlayer The SoundPlayer class is responsible for smooth playback by providing a constant data stream to the sound player. The method `playFromJitterBuffer()` collects one packet per call from the JitterBuffer. It controls its validity, and according to the outcome of this control, it chooses to either play the packet, play error correction data or play data corresponding to silence. If the JitterBuffer is empty, it plays either error correction data or silence. The only alternative to this behaviour is the very first call in a voice session, where the call to `playFromJitterBuffer()`

fills the whole playout buffer at once. This logic is shown in the activity graph in Fig.23.

Class VoiceActivityDetector The VoiceActivityDetector class contains the logic to decide whether a recorded ADB contains voice or silence, i.e. whether it should be sent or not.

Class UdpConnection This class creates an UDP socket connection with the other communicating end, and provides helper methods for socket handling.

Class JitterBuffer This class wraps around a priority queue. It's purpose is to smooth jitter and to order packets, should they arrive out of order. It is normally placed between the RtpReceiver and the SoundPlayer.

Class Audiobuffer This class wraps around a queue. It's purpose is to provide buffer space between the SoundRecorder and the RtpSender. It is particularly useful if the SoundRecorder and the RtpSender run in different threads.

Class Properties This helper class reads configuration parameters for the voice component from a file and provides the parameter values to the other classes in the same package.

Class Logger This helper class provides logging services to the classes in the package.

Collaboration Diagram

This diagram, shown in Fig.17, illustrates how the objects communicating. The thread objects are stereotyped.

7.2.3 Behavioural Diagrams

Use Case Diagram

The use case diagram is shown in Fig.18. It is set on the level of the users of the application.

Controller Behaviour

The activity graph in Fig.19 illustrates how the tasks of the main controller are split up on several threads to optimize scheduling policies. Note: some activities are modelled as activity-states. This is done when a waiting state is contained in the flow. Creation and termination of the controller threads is shown in Fig.20.

Record and Send Behaviour

The main record- and send behaviour is shown in Fig.21. Recording is started with the `start()` call. The method `getADB()` returns as soon as an ADB is ready for processing in the recorder buffer. Then, voice activity detection is performed on the ADB. If voice is detected, the ADB is inserted into the AudioBuffer, from where it is accessed by the RtpSender. It is then wrapped into an RTP packet and sent to the peer. In our design, the SoundRecorder and the RtpSender work in sequence, controlled by the recorderSendCtrl thread.

Receive Behaviour

The sequence of actions performed by the receiver is shown in Fig.22. In the beginning, the network socket is cleared of data that was sent by the peer before playback behaviour is started. This is necessary to minimize end-to-end delay. The `awaitNewData` waiting state returns as soon as data is available in the network socket.

Playout Behaviour

The playout behaviour is designed in three diagrams. The first is a refinement of the playFromJitterBuffer activity state that we have already seen in the activity graph of Fig.19[1]. This refinement, shown in Fig.23, is again an activity graph and shows the logic of how to decide what data to play. As a result of this decision, the SoundPlayer either fills the oufBuffer (in the very beginning of a voice session), plays the packet that was polled from the JitterBuffer (the normal behaviour), plays the last packet as error correction data or silence if the packet is not valid or if the JitterBuffer is empty.

In the beginning, the `playFromJitterBuffer` activity enters the `Initialbuffering` waiting state. It remains there until it gets notified by the JitterBuffer that the later can provide the SoundPlayer with enough data to fill its playout buffer. The playout buffer must be filled to its maximum to guarantee smooth playback. After this, timely re-invocation of the `playFromJitterBuffer()` method will guarantee that the load level is kept maximal. The first sequence of actions is also shown in the sequence diagram of Fig.24.

Finally, the main flow of the playout behaviour is shown in the sequence diagram in Fig.25. There is one waiting state in the sequence - `waitPlayDone`. Note: for performance reasons, it is important that this waiting state is outside the region between the `get()` call and the `waveOutWrite()` call. If it is within this region, it becomes possible that packets get rejected, even though they arrived on time.

[1]In this diagram, an arrow that leaves a decision node to the left or the right has the semantics "decision-statement evaluated to true"

7.2.4 Discussion

With this model we designed important aspects of the voice component. Note the annotations of the sequence diagrams to illustrate loop and and conditions, especially the one in Fig.21: with UML 2.0, the modelling language for sequence diagrams has been enhanced in a way to directly express such behaviour.

The language defined to annotate the activity diagrams should be expressive enough to allow easy communication of real-time constraints between humans. It is quite intuitive and not thought to be used by a tool to analyze real-time constraints. To make the language usable by a tool, it needs to be unambiguous, more specific and more expressive.

#ref	TimeObservationAction 1	TimeObservationAction 2	TimeConstraint	Description
1	initializeRecordSend.enter	getToAudioBuffer.wait.enter	ADB.playtime; soft	The time span between and start of recording and the first packet polling from the recorder queue should be < ADB.playtime. This is a soft real-time constraint; Isolated transgressions of this constraint critical.
2	getToAudioBuffer.wait.leave	getToAudioBuffer.wait.leave	ADB.playtime; soft	`getToAudioBuffer.wait.leave` should be revisited at least every `ADB.playtime`. In other words, recorded packets should be taken from the recorder queue as soon as they are recorded. This is a soft real-time constraints.
3	receiveToJitterBuffer.wait.leave	receiveToJitterBuffer.wait.leave	ADB.playtime; soft	A receive-action should be executed at least every ADB.playtime. This is a soft real-time constraint. Transgressions are to be expected because of jitter.
4	playFromJitterBuffer.wait.leave	playFromJitterBuffer.wait.leave	ADB.playtime; hard	An ADB must be scheduled for playback exactly every. This is a hard real-time constraint.

Table 14: Real-Time constraints for the voice component.

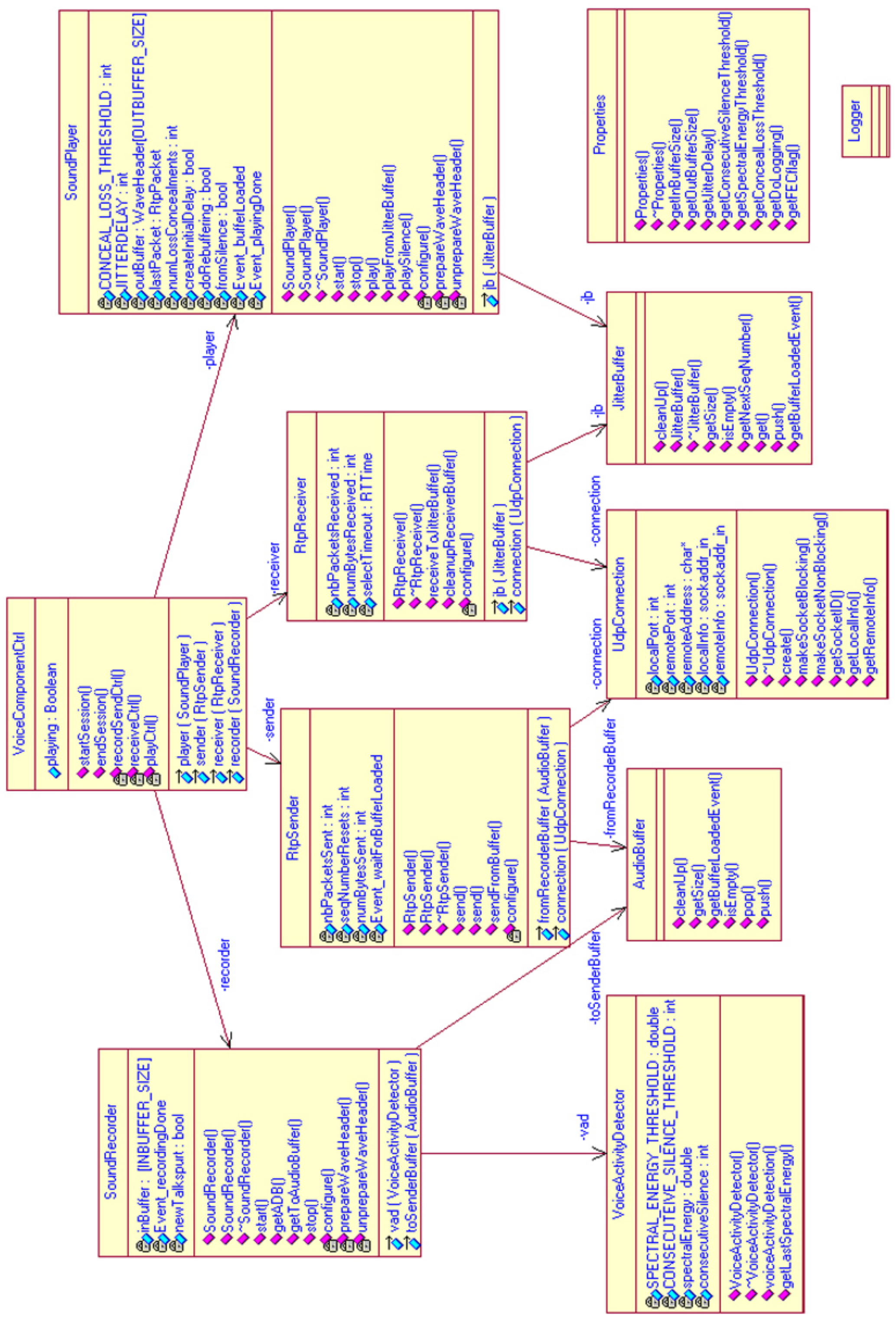


Figure 16: The class diagram.

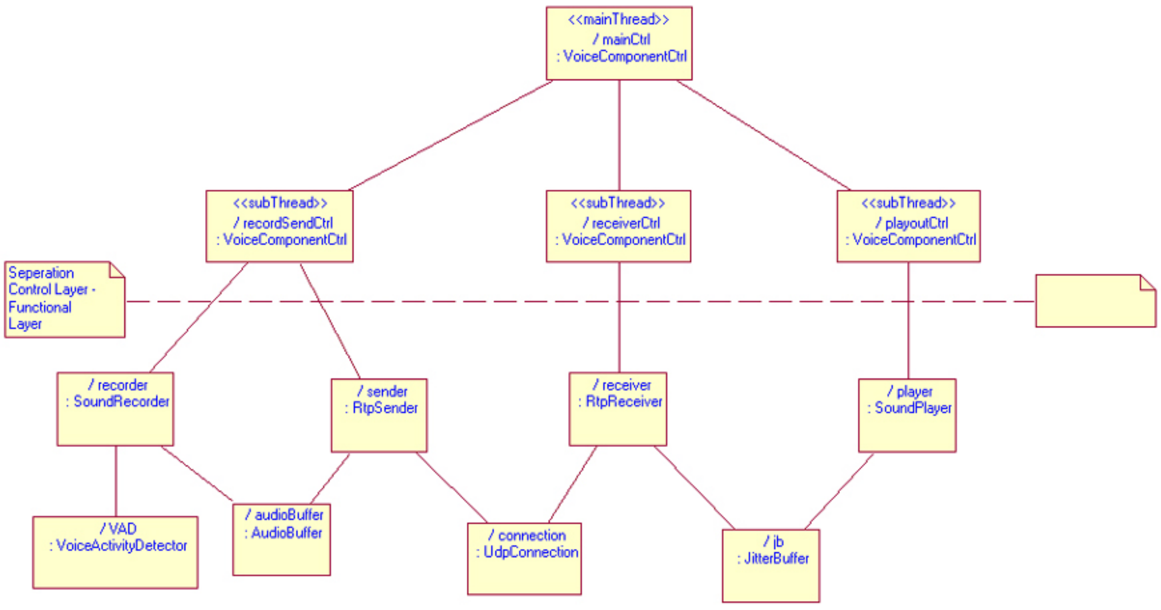


Figure 17: The collaboration diagram.

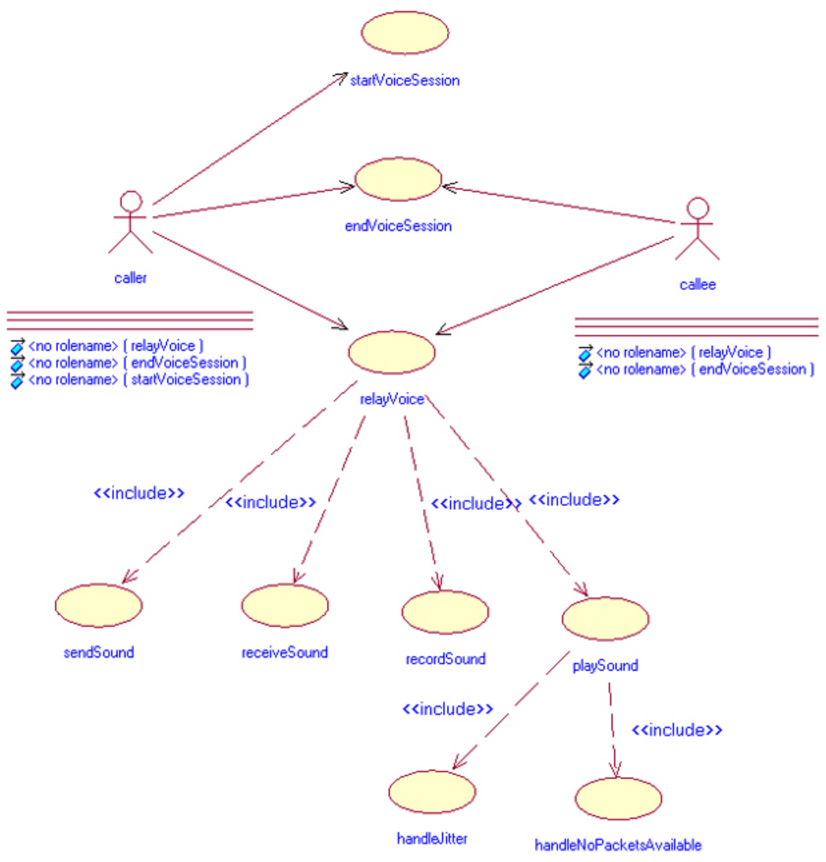


Figure 18: The use case diagram for the voice component.

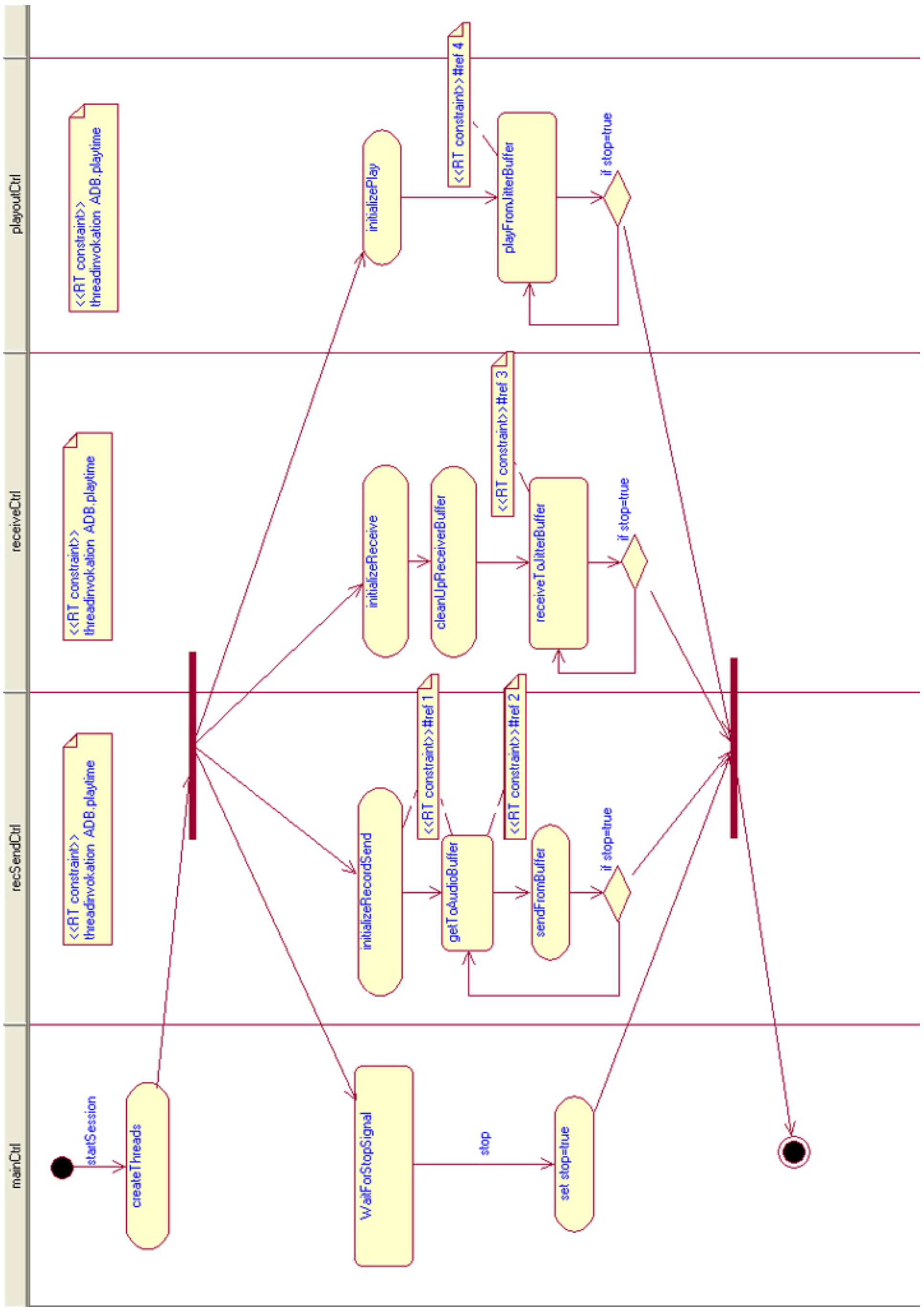


Figure 19: Activity Diagram for the control threads.

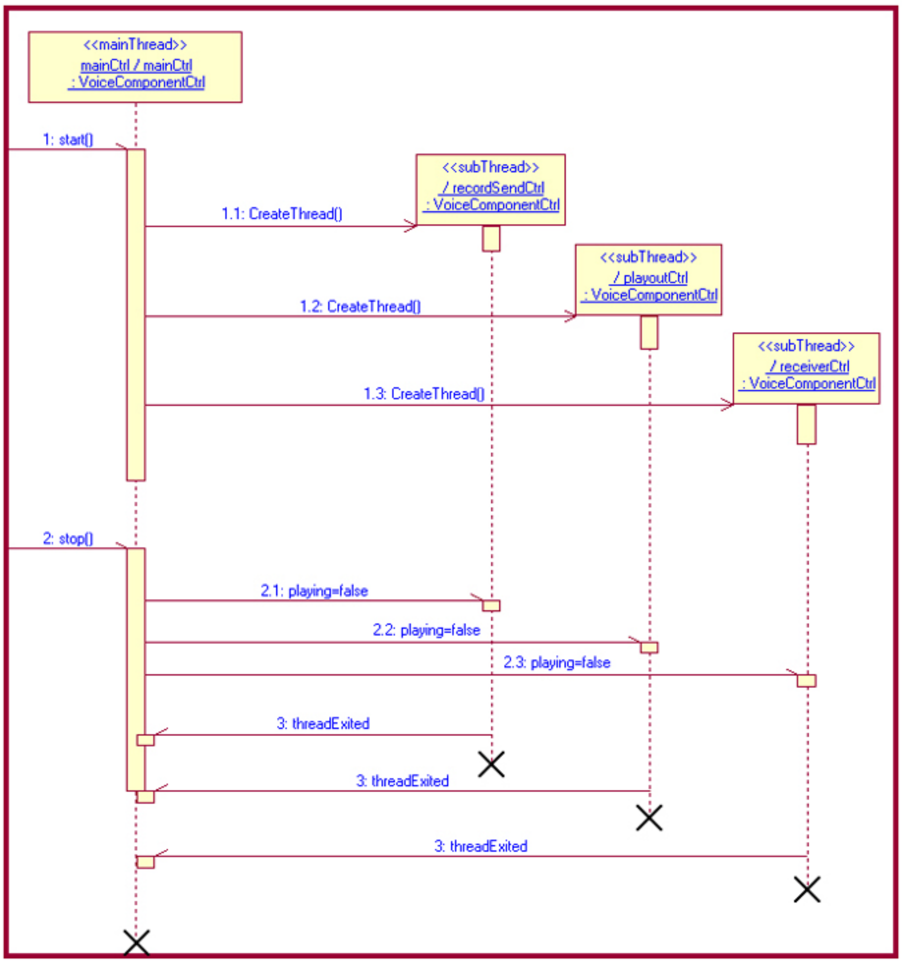


Figure 20: Creation and termination of threads on user inputs (start/stop).

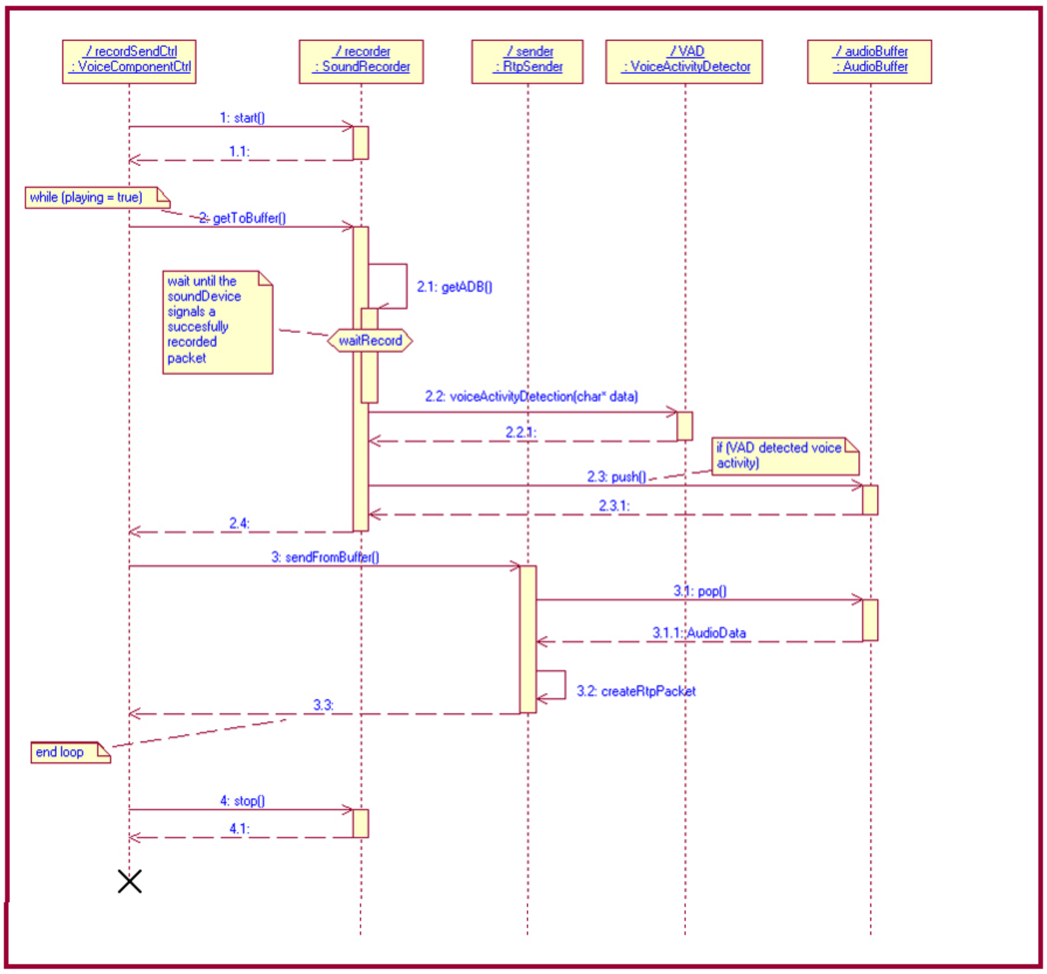

/ recordSendCtrl : VoiceComponentCtrl
/ recorder : SoundRecorder
/ sender : RtpSender
/ VAD : VoiceActivityDetector
/ audioBuffer : AudioBuffer
1: start()
1.1:
while (playing = true)
2: getToBuffer()
2.1: getADB()
wait until the soundDevice signals a succesfully recorded packet
waitRecord
2.2: voiceActivityDetection(char* data)
2.2.1:
if (VAD detected voice activity)
2.3: push()
2.3.1:
2.4:
3: sendFromBuffer()
3.1: pop()
3.1.1: AudioData
3.2: createRtpPacket
3.3:
end loop
4: stop()
4.1:

Figure 21: The main flow of the record- and send behaviour.

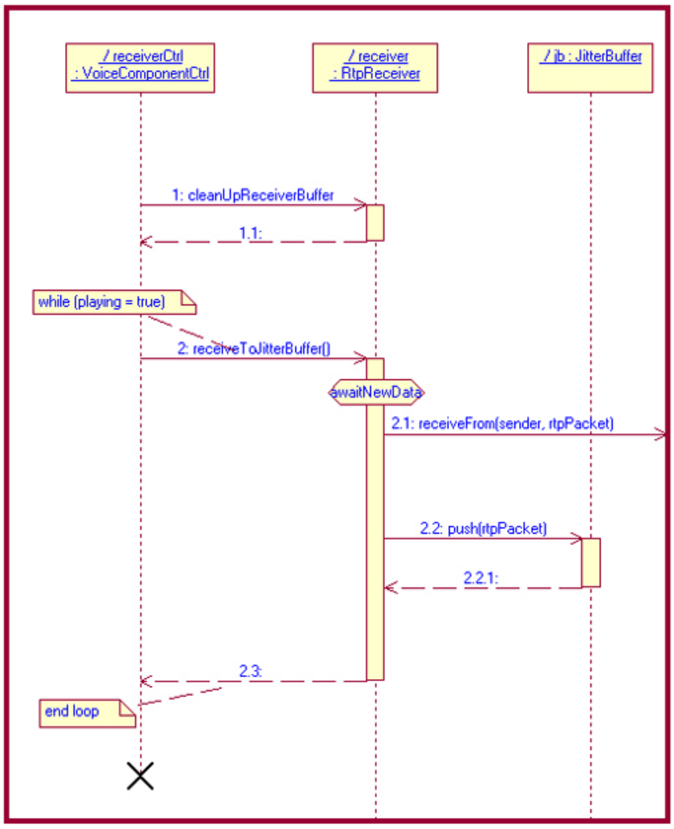


Figure 22: Main flow of receiver actions.

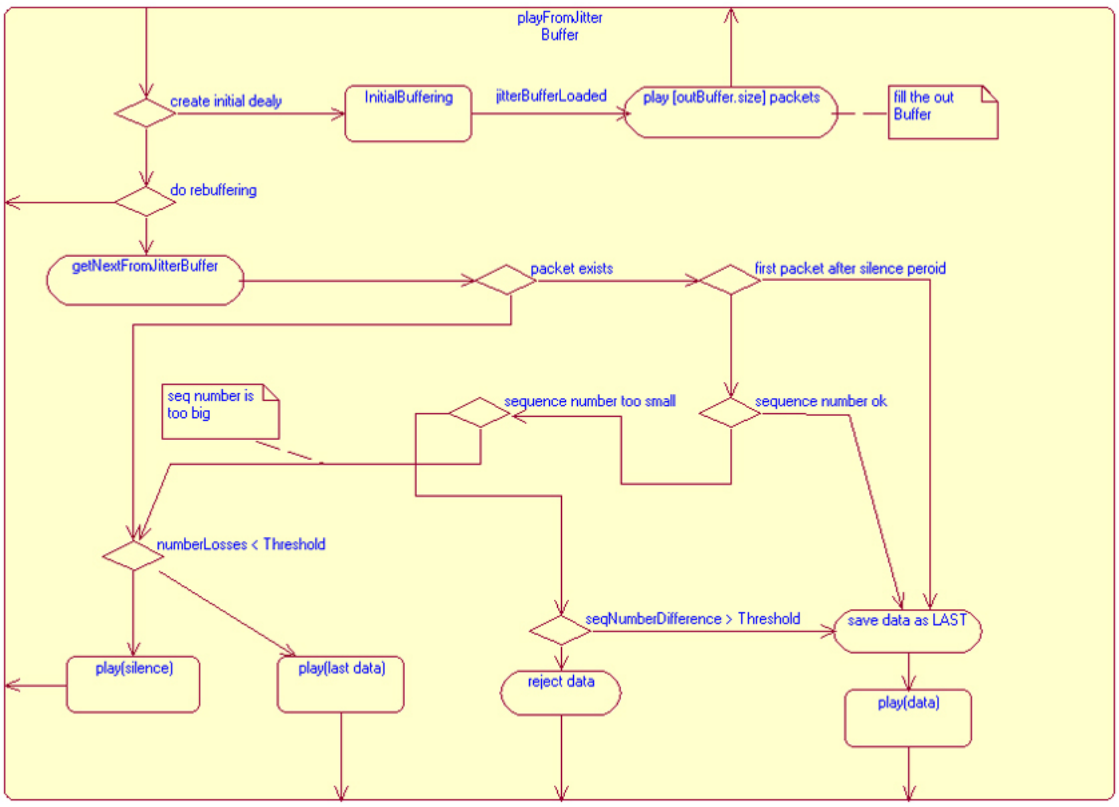


Figure 23: Details of the playFromJitterBuffer activity state.

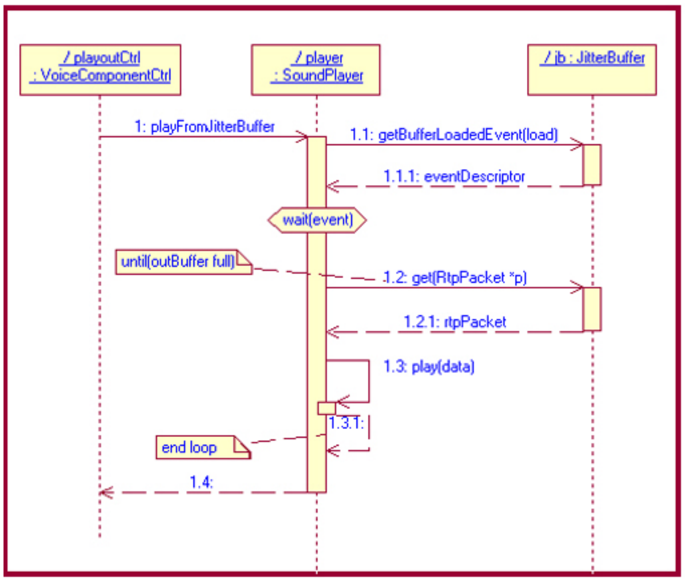


Figure 24: Alternative flow for the playout behaviour in beginning of a voice session.

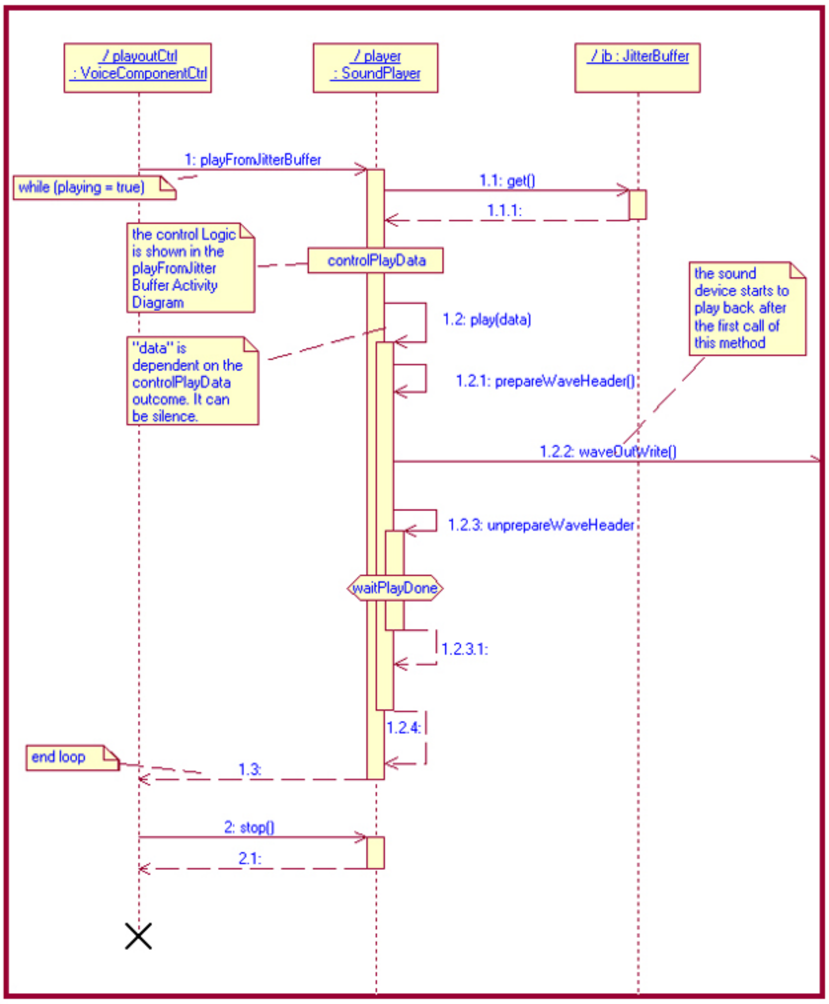

/ playoutCtrl : VoiceComponentCtrl
/ player : SoundPlayer
/ jb : JitterBuffer
1: playFromJitterBuffer
while (playing = true)
1.1: get()
1.1.1:
the control Logic is shown in the playFromJitter Buffer Activity Diagram
controlPlayData
the sound device starts to play back after the first call of this method
1.2: play(data)
"data" is dependent on the controlPlayData outcome. It can be silence.
1.2.1: prepareWaveHeader()
1.2.2: waveOutWrite()
1.2.3: unprepareWaveHeader
waitPlayDone
1.2.3.1:
1.2.4:
end loop
1.3:
2: stop()
2.1:

Figure 25: Main flow for the playout behaviour.

Chapter 8

Implementation

In this chapter we describe how the voice component is implemented in different classes with a focus on how these classes are used within our scope. The general use of all classes is described in the voice component API in Appendix B.

8.1 The Voice Component Package

8.1.1 Class VoiceComponentControl

This is the only class that should be accessed by the application that uses the voice component. Its two methods, `startSession()` and `endSession()` start and stop voice streaming.

After the start signal is received, the class instantiates all functional components. It then creates 3 threads: One to control recording and sending (which run in sequence), one to control receiving and one to control playback. The threads are created using the Win32 ATL `CreateThread()` command. The command must be provided with the name of the thread procedure of the new thread. In our case this corresponds to one of the three control procedures.

Every thread has one waiting state. The RecorderSender thread waits if no full ADB is available in the recorder buffer. The Receiver thread waits until new data arrives on the network. The player thread waits until an ADB has finished playback before it continues buffering. It waits *only* if there are no free WHs in the playout buffer. This set of waiting states is minimal and seems to be optimal in our configuration.

A strange runtime behaviour that is related to the multi-threaded implementation was observed. A user that runs the application must be registered as a local user on the machine, although minimal rights are enough. If not, WinXP complains that the user doesn't have enough rights to run the voice component. Whether this behaviour is a bug in WinXP, the normal behaviour of WinXP or a bug related to our implementation of threads (which is done following a standard approach) is unclear and not further investigated in the scope of this work.

The class is static because the threading API is designed for a procedural environment.

8.1.2 Class SoundRecorder

Sound recording is done using the Win32 API for sound. The following listing shows the audio capturing logic of the class: the method `getADB()`.

```
WAVEHEADER recorderBuffer[]; SOUNDEVENT eventRecDone; ADB data;
int recorderBufferIndex = 0;

ADB getADB(){

    if(recorderBuffer[recorderBufferIndex].flag != FINISHED_RECORDING){
        WaitFor(eventRecDone)
    }
    data = recorderBuffer[recorderBufferIndex].data;

    unprepareWaveHeader(recorderBufferIndex);
    prepareWaveHeader(recorderBufferIndex);

    recorderBufferIndex = (recorderBufferIndex+1) mod recorderBuffer.size;

    return data;
}
```

Listing 8.1: pseudo code for method getADB()

1. If, and only if, the WH described by `recorderBufferIndex` is still in recording state, wait until it has finished recording.
2. The ADB of the WH is copied in a temporary location.
3. The WH is unprepared (cleared) and then reprepared.
4. A counter is incremented to point to the next WH in the RQ
5. The copied ADB is returned to the calling entity.

If a recorded WH is already waiting, this method doesn't enter a waiting state. After having recorded one ADB, voice activity detection is performed on the data. If voice is detected, the packet is wrapped into a container and inserted into an AudioBuffer (see sect. 8.1.9). If the ADB is the first one in a new talkspurt, the container is marked. In our implementation, this container already has the structure of an RTP packet, and the mark corresponds to setting the header field `m` to `1`.

Configuration Parameters

This class uses the configuration parameter *InBufferSize*. It describes the size of the In Buffer in ADBs. It should be big enough to hold all recorded packets. This size of this buffer is stressed when the sending-thread is interrupted by the operating system. The value for this parameter can be large, as it has no direct effect on end-to-end delay.

8.1.3 Class RtpSender

The RtpSender takes a packet from the AudioBuffer. As this packet already has the RTP structure, all it needs to do is set the remaining header fields: the sequence number. The sequence number corresponds to the number of packets that have been sent before this packet. If the sequence number exceeds the value 65000, it is reset to 0. The packet is then sent over the network to the peer defined by an UdpConnection (see section 8.1.7).

8.1.4 Class RtpReceiver

This class takes RTP Packets from the network and puts them into a JitterBuffer. This buffer also functions as a communication channel with the SoundPlayer. In order to get notified at the arrival of new packets, the `select()` command with a timeout is used. A timeout is necessary to check periodically if the voice session is still active, or if the controller has given the signal to exit this thread.

Normally, one of the participants starts recording and sending data before the other participant starts playing back. The result is that data gets accumulated in the socket of the receiver. This data must be cleared in the beginning of a voice session, otherwise a delay is created. There might be data lying in the socket that was recorded and sent several seconds ago. The `cleanupReceiverBuffer()` method is used for this purpose.

Configuration Parameters

select timeout: 500ms.

8.1.5 Class SoundPlayer

The main objective of this class is to keep its playout buffer as full as possible. This is achieved by buffering voice data if the call participant is in a talkspurt, or silence substitution data otherwise. It also implements the *repetition* error correction mechanism.

Aside: Select(), timeouts and file descriptor sets.

The sockets that need to be observed are given to the `select()` command via a file descriptor set, FD_SET. If a timeout occurs and none of the sockets in the FD_SET showed activity, then the sockets are *removed* from this set. It is thus necessary to re-add the RTP socket to the set at every invocation of the method. This is a behaviour defined by the Berkeley sockets, on which Windows sockets are built. It is not documented in the Windows socket API documentation.

The listing for the method interacting with the sound-device, `play()`, is given below:

```
WAVEHEADER playoutBuffer[];
ADB data;
int playoutBufferIndex=0;

play(data){
    prepareWaveHeader(playoutBufferIndex, data);
    waveOutWrite(playoutBuffer[playoutBufferIndex]);

    playoutBufferIndex = (playoutBufferIndex + 1) mod playoutBuffer.size;
    unprepareWaveHeader(playoutBufferIndex);
}
```

Listing 8.2: pseudo code for method play()

1. Prepare the WH for the playback of `data`. The WH described by `playoutBufferIndex` is *not* currently playing, thus no waiting state is entered.
2. Write a pointer to this WH to the sound device.
3. Increment playoutBufferIndex.
4. Unprepare the next WH. If the next WH is still playing, this method enters a waiting state until the WH has finished playing. When the method returns, an empty WH is available in the playout buffer.

The `play()` method is called in a loop, i.e. as soon as it exits, it is called again. If no data has arrived by the moment of the call, a substitute or silence is played. By putting the `unprepareWaveHeader()` statement at the *end* of the method, we are allowing new data a maximum of time to arrive. (In the old design, the `unprepareWaveHeader()` statement was in the beginning of the method.)

The logic of the `playFromJitterBuffer()` method is shown in Fig.23. As soon as the method has decided on what to play, it calls the `play()` method.
The `InitialBuffering` waiting state is left when the JitterBuffer has a load equal to *JitterDelay + OutBufferSize.* When it is left, the playout buffer and the jitter buffer instantly reach their ideal load. The delay introduced by this buffering scheme is thus proportional to *JitterDelay + OutBufferSize.* With our default configuration, this delay

Data format	Maximum value	Minimum value	Midpoint value
8-bit PCM	255 (0xFF)	0	128 (0x80)
16-bit PCM	32,767 (0x7FFF)	32,768 (0x8000)	0

Table 15: Numerical values for PCM samples

PCM Format	Description
8-bit mono	Each sample is 1 byte that corresponds to a single audio channel. Sample 1 is followed by samples 2, 3, 4, and so on.
16-bit mono	Each sample is 2 bytes. Sample 1 is followed by samples 2, 3, 4, and so on. For each sample, the first byte is the low-order byte of channel 0 and the second byte is the high-order byte of channel 0.

Table 16: Data packing for the 8/16 bit mono PCM waveform audio data

is 112ms. This corresponds to an *OutBufferSize* = 4, *JitterDelay* = 3, 1 ADB containing 128byte of 8bit/8000khz/mono PCM data.

Configuration Parameters

This class uses three configuration parameters.
ConsecutiveLossThreshold. This parameter decides after how many consecutively lost or error-corrected packets, a rebuffering of the jitter buffer should be initiated before continuing to play voice. It needs careful adjustment, and should be between 2 and 4. With bigger values, the application might get trapped in a receiving/discarding pattern: it receives packets just fast enough that this threshold doesn't catch, but all the received packets are arrive too late, so they are discarded. The result is that silence is played. On the other hand, the bigger the value for this parameter, the better error-correction becomes audible.
OutBufferSize. This parameter describes the size of the playout buffer, in ADDs. It is highly important for good voice quality. It should be big enough to allow the sound device to playback without glitches, but as small as possible in order to keep end-to-end delay minimal. It depends on the performance of the machine, sound-driver and soundcard. See Tab.11.
JitterDelay. This parameter describes the size of the Jitter Buffer, in RTP packets. This value depends on the quality of the link - it should be big enough to smooth jitter, but as small as possible in order to keep end-to-end delay small.

8.1.6 Class VoiceActivityDetector

The Energy of an ADB is calculated according to the following formula

$$E = \frac{1}{N} \sum_{i=1}^{N} x(i)^2$$

where E is the Energy, N is the number of samples and $x(i)$ is the value of sample i. The value for $x(i)$ is calculated according to the value range for PCM samples given in Tab.15 and their packing given in Tab.16.

$$x(i) = x_{8bit}(i) - 128$$
$$x(i) = x_{16bit}(i)$$

The voice component works with 8-bit mono PCM samples. For the sound data recorded on our main system we obtain good silence detection with $E_{th} \cong 3$.

A strange phenomenon was observed on our second test system: the values for samples that were recorded in the absence of voice activity (unplugged microphone) was not 128, but 124. Thus, for "silent" ADBs, we obtained an signal energy $E = 16$. By taking a threshold $E_{th} = 16.1$, VAD works well though.

Configuration Parameters

This class uses two configuration parameters.
SpectralEnergyThreshold E_{th}. E_{th} decides whether silence or voice activity is detected. It must be adjusted for each machine and soundcard on a trial and error basis. If its value is 0, VAD is disabled. We implemented a tool called `VAD Detection Utility.exe` that comes with the ChattaBox client software and that can be used for this purpose. See section 8.3.1.
ConsecutiveSilenceThreshold C_{th}. C_{th} is necessary because a talkspurt naturally contains silent periods. An ADB containing silence but lying in the middle of a talkspurt shouldn't be discarded. It is important important for good voice quality to keep them; otherwise glitches are audible in the voice stream. C_{th} determines after how many silent ADBs the VAD should declare that a silent period has started.

8.1.7 Class UdpConnection

This class establishes, manages and stops a UDP connection between two participants in a voice session.

8.1.8 Class JitterBuffer

This class can be used by a single thread and is at most used by 2 threads. If it is used by two threads, it provides a inter thread communication channel that arranges the contained data structures according their priority. It wraps around a the class template `priority_queue` provided by the Windows C++ SDK. It contains RTP packets. The packet with the lowest sequence number has the highest priority.

The `priority_queue` class is thread safe. Note that the implementation of the Jitter-Buffer reduces this thread safety. Consider a first object using the JitterBuffer that executes the following action sequence:

1. Enquire a property of the highest priority packet,
2. Pop the packet from the queue.

A second object using the same buffer might introduce, between action 1 and action 2, a new packet in the JitterBuffer that takes the highest priority. These cases are very rare, though. On the other hand, by introducing synchronization structures such as a CriticalSection to obtain thread safety, we slow down the responsiveness of the buffer, for reasons mentioned in section 4.4.1: Synchronization mechanisms rely on event notification in order to find out when they can enter a section that is blocked, and these events do not always arrive as fast as we would like.

We gave buffer access speed a higher priority than isolated decision errors due to wrong information on the highest priority packet.

8.1.9 Class AudioBuffer

This class is very similar to the JitterBuffer. It provides a FIFO queue between two objects. It is thread safe.

8.1.10 Class Logger

This class provides logging services, that may or may not be timestamped. For time stamping, the Windows `GetSystemTime(&SYSTEMTIME)` command is used and has a resolution of 1 ms.

Configuration Parameters

This class uses the configuration parameter *doLogging*. If it is set to `1`, logs are created. If set to `0`, logging is suppressed.

No.	Name	Used by; described in section	Suggested Value
1	SignalEnergy Threshold	VoiceActivityDetector; see section 8.1.6	?
2	ConsecutiveSilence Threshold	VoiceActivityDetector; see section8.1.6	20
3	ConcealLoss Threshold	SoundPlayer; see section8.1.5	2
4	OutBufferSize	SoundPlayer; see section8.1.5	4
5	JitterDelay	SoundPlayer; see section8.1.5	3
6	InBufferSize	SoundRecorder;see section8.1.2	20
7	Logging	Logger; see section 8.1.10	0
8	FEC	RtpSender, RtpReceiver. Flag indicating whether Forward Error Correction should be applied or not. Must be the same on both communicating ends. 1 = yes 0 = no [is not implemented with this version of the Voice Component]	0

Table 17: The configuration parameters of the rtp.conf file

8.1.11 Class Properties

The voice component can now be configured via a configuration file called "rtp.conf". The Properties class reads the first line of this file and provides getters for these configuration values.

8.1.12 Type Definitions: RtpHeader.h

This header file defines important types, such as the RtpHeader and RtpPacket structure.

8.2 Configuration of the Voice Component

The configuration parameters for the voice component are found in two places. The rtp.conf file contains parameters that need to / can be adjusted from environment to environment. The existence of this file with values for each parameter is *obligatory.* Beside the rtp.conf file, the Constants.h C++ header file contains parameters that are compiled into the program.

SAMPLING_RATE	8000 (Hz)
SAMPLING_RESOLUTION	8 (bits/sample)
ADB_SIZE	128 (bytes)
RTP_PAYLOAD_SIZE	128 (bytes)

Table 18: Hard coded configuration parameters in Constants.h

8.2.1 The File rtp.conf

All the 8 configuration values are found on the first line. They must be separated by white space. The configuration values are described in Tab.17. They are ordered in the same way as in the file.

8.2.2 The File Constants.h

This file defines the configuration parameters shown in Tab.18. These parameters are hard coded on purpose; they shouldn't be changed without very careful consideration.

8.3 Command Line Utilities

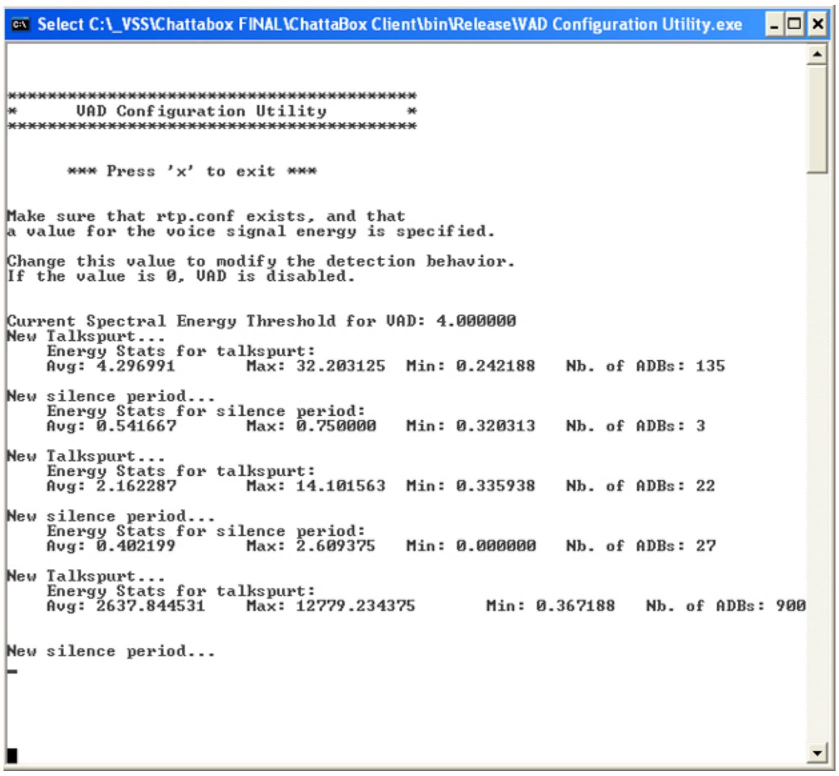
These utilities can be used from the DOS command line. They are very simple but useful to configure/test the voice component of ChattaBox.

8.3.1 VAD Configuration Utility.exe

This utility simulates a voice conversation with voice activity detection. It loops voice recorded at the microphone back to the earphone. The classes involved are the SoundRecorder, SoundPlayer and a JitterBuffer that links the two components.

The utility prints out a statement every time a silence/voice period starts, and some statistics every time a silence/voice period ends. It expects to find the rtp.conf file in the same directory.

The tool has primarily been developed to test the VAD behaviour. If the behaviour is not as required, the Signal Energy Threshold value in the rtp.conf file should be changed. In order to obtain a good overall voice quality, silence detection is important. A good VAD behaviour qualifies itself through extended silence periods, but talkspurts should not be "cut" in the beginning or the end. Basically, if nothing is said, silence should be detected. It is also a good idea to enable logging for configuring the Signal Energy Threshold parameter. In the logs for the SoundRecorder Class, the energy values and the decisions made by the VAD are logged on



Select C:_VSS\Chattabox FINAL\ChattaBox Client\bin\Release\VAD Configuration Utility.exe

```
******************************************
*      VAD Configuration Utility        *
******************************************


      *** Press 'x' to exit ***


Make sure that rtp.conf exists, and that
a value for the voice signal energy is specified.

Change this value to modify the detection behavior.
If the value is 0, VAD is disabled.


Current Spectral Energy Threshold for VAD: 4.000000
New Talkspurt...
    Energy Stats for talkspurt:
    Avg: 4.296991        Max: 32.203125  Min: 0.242188   Nb. of ADBs: 135

New silence period...
    Energy Stats for silence period:
    Avg: 0.541667        Max: 0.750000   Min: 0.320313   Nb. of ADBs: 3

New Talkspurt...
    Energy Stats for talkspurt:
    Avg: 2.162287        Max: 14.101563  Min: 0.335938   Nb. of ADBs: 22

New silence period...
    Energy Stats for silence period:
    Avg: 0.402199        Max: 2.609375   Min: 0.000000   Nb. of ADBs: 27

New Talkspurt...
    Energy Stats for talkspurt:
    Avg: 2637.844531     Max: 12779.234375       Min: 0.367188   Nb. of ADBs: 900


New silence period...
-
```

Figure 26: Screenshot of the VAD Configuration Utility.

The tool is generally useful to find out whether bad voice quality is due to the network or due to a wrong configuration of the voice component (such as the `playoutBufferSize`, etc.)

8.3.2 LAN Phone.exe

This utility allows easy setup of a voice communication with another computer. Both computers need to simultaneously run the tool in order for communication to be possible. No signalling (i.e. communicating to the other side when it has to answer a call) is used; signalling must be provided by other means. The tool is useful for testing the voice component and its configuration outside the context of the rest of the ChattaBox application.

8.4 Conclusion

The voice component described above has, with the proposed configuration, an end-to-end delay that is smaller than 150ms in the targeted environment (LAN). This value has been obtained the following way:

packetizing delay	<16ms	worst case waiting time for recording one ADB
sending delay	< 1ms	measured by `sendto()` method return-delay
network delay	< 1ms	measured with the `ping` command
reception delay	<16ms	maximum delay for enabled `select()` to return with WinXP (periodic clock)
jitter delay	48ms	corresponds to the playtime of 3 ADBs (jitter buffer size)
playout delay	64ms	corresponds to the playtime of 4 ADBs (playout buffer size)
end-to-end delay	**<146ms**	

The application handles buffers and waiting states in an optimal way. Delays are minimal in our hardware and operating system environment, except from the jitter delay, which we designed generously for operation in a LAN. The only way to further reduce delay would be to introduce asynchronous timer events on an operating system level and to work with a high performance sound driver and device.

Application tests with several users yielded that the current end-to-end delay is small enough to not be perceived within a conversation.

The voice component runs stable on long calls, i.e. without degradation in performance. Memory consumption is constant. This was checked with the Windows Task Manager by observing system resources for calls exceeding one hour.

The network load has been reduced by around 70%. The previous version only run with a 16bit sampling resolution. 70% reduction is obtained by making it run with 8bit (50% reduction) and applying VAD (another 30 to 50% reduction).

Although the network load has been reduced, the application is designed to run in an environment where relatively high bandwidth is available. If it is in a talkspurt, it generates data with a rate of 74kbps[1]. To make the application ready for a use over the Internet, the implementation of a specialized VoIP codec is necessary. One such codec is the G729.a codec, which operates at 8kbps.

Again, we would like to emphasize the importance of voice activity detection. It not only cuts down bandwidth use, but is essential for the component to run smoothly and without performance degradation over a long time. Performance degradations *do* occur within a talkspurt , i.e the delay increases. For this to be perceived, the sender must be constantly sending data for a duration of the order of minutes. As the duration of a typical talkspurt is in the order of seconds, the delay degradations stay unnoticeable. At the end of a talkspurt, the buffers at the receiver are played out, and therefore the accumulated delay is reset.

[1]In one second, 62.5 packets are sent, each containing 1024bits(128 bytes) of payload and a 160bits(20bytes) RTP-header

Chapter 9

Solution Verification and Final Conclusion

9.1 Solution Verification

The engineering objectives for this project were to reduce the end-to-end communication delay of the ChattaBox application to a value smaller than 200ms and to guarantee high voice quality for the duration of the call. This was to be achieved in the context of an extended Local Area Network as found in the Computer Science Department at the University of Cape Town. ChattaBox was presented to the DNA Group and other potential users. The improvements were clearly perceived and user acceptance was high. Several testers suggested deploying the application on the LAN. Thus, **the engineering objectives for this project have been achieved.**

The following engineering related achievements have been made:

- In the requested environment (LAN), the application runs with an end-to-end delay below 150ms (see 4.6). Voice quality is high for the duration of extended calls. The application is stable and without memory leak.
- The delay generated by the application is minimal and can only be further reduced if a different sound hardware, different sound driver and/or a different operating system is used.
- A Voice Activity Detection algorithm has been implemented. VAD is essential to maintain voice quality over time, and reduces network load.
- Network load has been reduced by approximately 70%.
- The bugs related to voice-mail recording and call termination have been resolved. They were found during the user level analysis of ChattaBox and appear in Tab.4.

- Important parameters of the application are configurable at runtime through a configuration file.
- The code-base of the voice component is better structured, easier to read and well documented, thus favoring further development of the component, i.e. adapting it to use over the Internet.
- Two command-line utilities have been developed that help the user obtain an optimal configuration of the voice component.
- A catalogue of general design recommendations for voice streaming over IP Networks was established .

The other objective for this project was to re-engineer the voice component using UML and to investigate how UML and UML base CASE tools can be used to support the re-engineering process. **The UML focused objective has been achieved.**

- Analyzed how effective state of the art real-time UML tools apply to the real-time domain.
- Established a comparison between UML 1.x and UML 2.0, and identified new concepts that can be helpful for the real-time engineer.
- Demonstrated how a UML activity graph can be annotated with real-time constraints, inspired by notions found in UML 2.0.
- Established a review of current research trends on the use of UML in a real-time environment.

9.1.1 Future Work

The integration of a *new codec* should be the main goal of a future development iteration. The G729 codec family should be investigated for this purpose. The G729.a package comes with its own VAD and error correction algorithms, which can be advantageous. An alternative is to use a codec that is already implemented in the sound device. This would save significant calculation time.

Implementation of the *RTCP protocol* is another important improvement. Advanced services such as conference calls rely on the information distributed by this protocol.

As soon as the application is to be used over a network with varying network throughput and delays, such as the Internet, the integration of an *adaptive jitter buffer* should be considered.

Enhancements for the remainder of the application have been suggested in Chapter 2.

9.2 Project Development and Management

The project was realized in two phases. In the first phase, we thought that optimizing the voice component was a small part of a bigger project. We finally realized that the related issues are complicated and that ameliorating the voice component was to be the core of this project. Project scope reorientation took one week (after the midterm presentation), in which we negotiated and produced a new project proposal accepted by the local research laboratory. The sub-phases associated with the first project phase are shown in Tab.19.

In Phase 2 we decided to resort to proven design methods and CASE tools so that the solution could be approached from a different, more abstracted view.

We first studied the case tool RoseRT, and realized that it did not meet our requirements for a real-time CASE tool. Thereafter, we decided to step back for a moment from our quest for a solution to the voice quality problem and spend a considerable amount of time on writing up what we had learned. Our goal was to get a grasp on the amount of time that we have left to investigate other developing methods, and to structure our thoughts and ideas. This sub-phase finalized, we made the following decision appropriate to the small amount of time left (6 weeks): spend a few days on investigating whether TauG2 can handle real-time constraints. If yes, model the application with TauG2. If not, model it with RoseRT in "normal" UML. The final implementation model must be established and implementation must be started *as soon as possible* due to the uncertain nature of the bug described in sub-phase 3 of the 1st project phase (playback with glitches). The deadline for implementation was set to the 13th of August, which would leave us three weeks for finishing the report. Finally, less time than scheduled was spent on implementation, and UML 2.0 could be studied instead, which turned out to be a very interesting part of our project. The sub-phases of the 2nd project phase are shown in Tab.20.

9.3 Self-Assessment

During this project, the following personal achievements have been made:

- Successfully identified relevant issues influencing voice quality in packet switched networks, and ways how they can be solved.
- Analyzed bugs by applying problem domain separation. This way, even hard to detect bugs (operating system/hardware level) could be found and their reasons analysed.
- Applied signal processing knowledge to implement the voice activity detector.

Sub-phase	Time Period	Issue	Result
1.	April	User level analysis of ChattaBox, make an assessment of its weaknesses. Prioritize the discovered issues.	Improvement of voice quality has highest issue
2.	April	Study VoIP and how to guarantee voice quality in packet oriented networks. In parallel, low level analysis of voice component	First design suggestions on how the voice component needs to be improved. Multi-thread idea, error correction, jitter-buffer.
3.	April - End of May	Improve the existing code base. Jitter buffer, multi-thread, error correction.	”Hacking” of the existing code base turned out to be time consuming, as code is not well structured and hard to read. Flaws have been detected and corrected in all parts of the system. Persisting problem at the SoundPlayer: even though an playout buffer with at least double buffering is guaranteed at the player, and data is enqueued all the time, playout has glitches and shows strange behaviours.
4.	26. May	Midterm presentation	Suggestion of my project supervisor to change project scope and focus on voice component.
5.	28. May - 4. June	Project scope reorientation	New (and final) project Proposal: ”UML in a Real-Time Environment: A Case Study”

Table 19: The sub-phases of project phase 1.

Sub-phase	Time Period	Issue	Result
1.	4. - 24 June	Learn RoseRT, come up with an executable model	Simple executable model of the voice component. Realized that RoseRT is not apt to model and verify real-time constraints.
2	24. - 30.June	Holidays	Decided that deliverables must be produced.
3.	1. - 18. July	Write background chapters on ChattaBox, VoIP, RoseRT	Some VoIP concepts clarified. Very clear idea on the VoIP parameters of the design. Half of the thesis is written.
5.	18. - 24.July	Study whether TauG2 can handle real-time constraints. Come up with an final implementation model	Final model is standard UML, as TauG2 cannot handle real-time constraints neither.
6.	25. July - 8. August	Implementation of the voice component according the final model.	Successful multi-thread implementation. Elimination of remaining bugs by identifying them to be operating-system/hardware related (time-inaccurate event notification, recording "faster" than playback). Voice Activity Detector implemented. Command Line tools.
4.	8.August - 3. September	Study UML 2.0 and the Profile for Schedulability, Performance and Time. Finish report	UML 2.0 and the Profile for SPT have many features that are interesting for the real-time engineer.

Table 20: The sub-phases of project phase 2.

- Applied a strict time-boxing approach in the second half of the project.
- Adapted well to a new programming language (C++).
- Successfully implemented communicating high-priority threads with the Win32 API. Got proficient with the Windows Sound API.
- As one of the first user of RoseRT in South Africa, gave a tutorial and live presentation of this CASE tool to the general distributor of Rational products in South Africa, Software Futures[1], with positive feedback.
- The project was successfully completed.

The following points could have been managed better in this project. They describe also general suggestions for a future project of this type.

- Background chapters should be written while background knowledge is acquired. This favors a clearer understanding of the intellectual material that is studied, and therefore early identification of relevant issues. In our project, background chapters should have been written before starting the first implementation iteration.
- A project management approach in *time-boxes* should be applied from the earl beginning. This includes establishment of a project plan. Time-boxing was applied in a strict manner only in phase 2 of the project.
- All along the project, a regular (weekly?) project- and self-assessment should be made. These assessments should serve mainly to verify that the project is heading in the right direction, and that the amount of time spent for a particular task corresponds to its designated time-box. *This should particularly be applied in times when the project advances slowly, and hard-to-solve problems are encountered.* We guess that we would have spent less time in the (unsuccessful) first implementation attempt if regular, thorough project assessments would have been made. Project- and self-assessments were better realized in the phase 2 of the project.
- A logbook should be established and be maintained during the *whole* project on a daily basis. It's useful to observe personal progress, and valuable as a documentation on what problems have been encountered/how much time was spent to solve them.
- Well defined project objectives should be established as soon as possible. For this, the opinion of experts in the area should be requested at early stages of the project, in order to obtain a realistic assessment of the situation. This was neglected during phase 1 of our project.

[1] www.softwarefutures.com

9.4 Conclusion

In this diploma project we successfully re-engineered and significantly improved the most important part of a Voice over IP application - its voice component. For this purpose, we first explained the problems that are inherent to voice streaming in packet switched networks - Jitter, Delay and Packet Loss - and how they can be addressed. We identified that Delay is the major problem that is encountered in our environment (LAN). We then showed the flaws of the initial voice component and pointed out that optimal data scheduling and optimal handling of waiting states are crucial for obtaining good voice quality. Furthermore, we showed that the real-time characteristics of WindowsXP is a bound to the minimum achievable delay because of the absence of asynchronous events. We showed that inaccurate time handling on a hardware level makes voice activity detection essential to maintain a good voice quality, specifically in order to keep delay low.

Furthermore, the usefulness of two popular Real-Time UML CASE tools have been studied in the context of our re-engineering iteration. It was shown that the notion of time is not an inherent concept, and that the tools do not meet our requirements for a real-time case tool. Emerging standards such as UML 2.0 and the UML profile for Schedulability, Performance and Time were examined to get an idea in how they improve/extend UML 1.x. Our opinion is that both standards are promising, that UML 2.0 greatly improves the modeling concepts available to the software engineer, and that a combination of UML 2.0 with an adapted Profile for SPT has the potential to significantly improve real-time application modelling.

9.4.1 Personal Conclusion

One of the exciting things about this project was that it allowed me to learn about a broad range of topics, ranging from the handling of timer interrupts in Windows XP to the novelties in UML 2.0. Very interesting was also what I learned on project management and working methods, be this through supervisors, from the style the local research group is lead or through personal experience. All in all, working for this thesis with the Data Networks Architecture Group was a very valuable, equally challenging and rewarding personal and professional experience.

Bibliography

[1] J. Landman, M. Marconi, M. Chetty, O. Ryndina *ChattaBox: A Case Study in UsingUML and SDL for Engineering Concurrent Communicating Software Systems.* October 2002. University of Cape Town. Department of Computer Science. B.Sc. Project Report. July 8 at http://www.cs.uct.ac.za/Research/DNA/ChattaBox/documents/report.pdf

[2] A. Muthukrishnan *An Empirical Study of VoIP.* July 2002. Indian Institute of Technology. Department of Electrical Engineering. M.Tech and B.Tech Thesis. July 4 at http://networks.ecse.rpi.edu/ anand/voip/reports/3stage/final_thesis.pdf

[3] C. Perkins, O. Hodson, V. Hardman *A Survey of Packet Loss Recovery Techniques for Streaming Audio* IEEE Trans. on Networking, pg. 40-48. September/October 1998. July 11 at http://csperkins.org/publications/IEEE-Network-1998.pdf

[4] Yi. J. Liang, N. Frber, and B. Girod *Adaptive Playout Scheduling and Loss Concealment for Voice Communication over IP Networks.* IEEE Trans. on Multimedia. to appear Dec. 2003. July 11 at http://www.stanford.edu/ bgirod/pdfs/LiangMM2002.pdf

[5] J. Ryan *Voice over IP (VoIP)* 1998. The Applied Technologies Group, Inc. White paper. July 8 at http://www.itpapers.com/techguide/voiceip.pdf

[6] S. Pracht, D. Hardman *Voice Quality (VQ) in Converging Telephony and IP Networks.* Electrical Design News, pg. 89-104. September 2000. July 8 at http://www.e-insite.net/ednmag/contents/images/47172.pdf

[7] *Voice over IP:Per Call Bandwidth Consumption.* Cisco Systems Inc. 2001. July 9 at http://www.cisilion.com/pdfs/Tech_VOIP_Bandwidth.pdf

[8] G. Booch, J. Rumbaugh, I. Jacobson *The Unified Modeling Language User Guide.* 1999. ISBN 0201571684. Addison-Wesley.

[9] B. Selic, J. Rumbaugh *Using UML for Modeling Complex Real-Time Systems.* March 1998. ObjecTime Limited/Rational Software Corp. White paper. July 8 at www.rational.com/products/whitepapers/UML-rt.pdf

[10] Object Management Group. *OMG UML 2.0 Superstructure Specification.* Final Adopted Specification, OMG Adopted Specification ptc/03-08-02, 2 August 2003.

[11] Object Management Group. *OMG UML 2.0 Superstructure RFP.* 3rd revised submission, OMG Document ad/03-04-01, 1 April 2003.

[12] Object Management Group. *UML Profile for Schedulability, Performance, and Time specification* OMG Adopted Specification ptc/02- 03-02, March 2002.

[13] Object Management Group. *Unified Modeling Language Specification, v. 2.0 Request For Proposal.* Draft. OMG Document: ad/2000-06-06

[14] Object Management Group. *OMG Unified Modeling Language Specifi- cation Version 1.5.* OMG Document formal/2003-03-01, March 2003, http://www.omg.org

[15] P. Leblanc *A stronger UML for Real-Time Development.* Embedded Systems Conference, San Jose, CA. Nov. 1999.

[16] L. Bichler, A. Radermacher, A. Schuerr *Evaluating UML Extensions for Modeling Real-time Systems.* Preceedings of the 7th Internationsl Workshop on OO Real-Time Dependable Systems (WORDS). IEEE Computer Society

[17] L. Lavazza, G. Quaroni, M. Venturelli. *Combining UML and formal notations for modelling real-time systems.* Proceedings of the 8th European software enguineering conference, 2001, Vienna, Austria. ACM Press, New York.

[18] M. Saksena, B. Selic *Real-Time Software Design - State of the Art and Future Challanges.*IEEE Canadian Review. June 1999.

[19] B. Selic, G. Gullekson, P.T. Ward. *Real-Time Object-Oriented Modeling.* ISBN: 0471599174, John Wiley & Sons. April 22, 1994.

[20] B. Selic, *Brass Bubbles: An Overview of UML 2.0 (and MDA)*Tutorial presented at OTS'2003 18-19 June 2003. Available 26 June 2003 at http://lisa.uni-mb.si/cot/ots2003/predkonferenca/.

[21] *Representations - Using UML/XMI and ADLs to Support System Adaptation* July 29 at http://www.csis.ul.ie/Research/aci/Documents/Representations.pdf

[22] Object Management Group, U2 Partners *UML 2.0: Superstructure, 3rd revised submission.* OMG Document ad/03-04-01. 1 April 2003. http://www.omg.org

[23] R. Pradhan (CASE digital), C. Szyperski (Microsoft), A. Taivalsaari (Sun Microsistems), A. Wills (TriReme) *Are Components Objects?* Summary of Panel Discussion, 29th of July at http://www.acm.org/sigplan/oopsla/oopsla99/2_ap/tech/2d1a_arecmp.html

[24] B. Møller-Pedersen *SDL Combined with UML.* April 2000. July 29 at http://www.item.ntnu.no/fag/SIE5020/UMLandSDL/ Telek4_2000%20SDL-UML.pdf

[25] B. Selic. *An Architectural Pattern for Real-time Control Software* In J.M. Vlissides, J.O. Coplien, and N.L. Kerth. (eds.) Pattern Languages of Program Design 2. Addison-Wesley, USA, 1996. 26 Aug at http://citeseer.nj.nec.com/selic96architectural.html

[26] ITU-T, Recommendation Z.100 *Specification and Description Language(SDL).* Geneva, 1996. http://www.itu.ch

[27] S. Graf, I. Ober *A Real-time Profile for UML and how to adapt it to SDL* 31 March 2003. VERIMAG. 2 July at http://www-omega.imag.fr/queries/dm-downloads.php

[28] J. Warmer *The future of UML.* March 2001. Klasse Objecten. 24 August at http://www.klasse.nl/english/uml/uml2.pdf

[29] *Guidelines For Providing Multimedia Timer Support*Microsoft Cooperation, Sept. 2002. August 20 at http://www.microsoft.com/whdc/hwdev/platform/proc/mm-timer_Print.mspx

[30] K. Ramamritham, Ch. Shen, O. Gonzales, S. Sen, S. Shirgurkari *Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations* 1998, IEEE Real Time Technology and Applications Symposium, pg. 102-111, August 20 at http://citeseer.nj.nec.com/ramamritham98using.html

[31] J. Grabowski, D. Hogrefe, G. Rethy, I. Schieferdecker, A. Wilese, C. Willcock *An Introduction into the Testing and Test Control Notation (TTCN-3)* Computer Networks, Volume 42, Issue 3, Elsevier, Amsterdam, 2003, 375-403. 25 August http://www.swe.informatik.uni-goettingen.de/pubs/single_pub/index.php?lang=de&pub_nr=225

[32] Z. R. Dai, J. Grabowski, H. Neukirchen *Timed TTCN-3 - A Real-Time Extension for TTCN-3.* International Conference on Testing of Communicating Systems March, 19th - 22nd, 2002. Berlin, Kluwer Academic Publishers, 2002, pp. 407-424. August 25 http://www.itm.uni-luebeck.de/pubs/single_pub/index.php?lang=en&pub_nr=183

[33] I. Schieferdecker, Z. R. Dai, J. Grabowski, A. Rennoch *The UML 2.0 Testing Profile and its Relation to TTCN-3.* Proceedings of the 15th IFIP International Conference on Testing of Communicating Systems (TestCom2003), Sophia Antipolis, France, May 2003. Lecture Notes in Computer Science (LNCS) 2644, Springer, 2003, pp. 79-94. August 25 at http://www.swe.informatik.uni-goettingen.de/pubs/single_pub/index.php?lang=de&pub_nr=223

[34] N. de Wet *Software Performance Engineering with UML-RT and UML 2.0* Literature review. University of Cape Town, Department of Computer Science. 25 August at http://people.cs.uct.ac.za/ ndewet/Performance_with_UML-RT.pdf

[35] G. Engels, J.M. Ḱuster. *Enhancing UML-RT Concepts for Behavioral Consistent Architectural Models* Proceedings of the 1st ICSE Workshop on Describing Software Architecture with UML, Toronto, Canada, May 2001

[36] K. Diethers, U. Goltz, S. Vocke. *Analysis of Real-Time Systems Modeled by UML-Statecharts* Proc. of the 1. Int. Colloquium of the Collaborative Research Centre 562, Braunschweig May 29-30, 2002, Fortschritte in der Robotik, Band 7, Shaker, Aachen, 2002. August 25 at http://www.cs.tu-bs.de/ips/diethers/publications.html

[37] L. Lavazza, G. Quaroni, M. Venturelli. *Combining UML and formal notations for modelling real-time systems* Proceedings of the 8th European software engineering conference,Vienna, Austria, Pages: 196 - 206, 2001. August 26 at http://portal.acm.org/citation.cfm?id=503236&coll=portal&dl=ACM&ret=1#Fulltext

[38] S. Flake, W. Mueller *Specification of Real-Time Properties for UML Models* Proceedings of 35th Annual Hawaii International Conference on System Sciences (HICSS'02), Volume 9, January 07-10 2002, Hawaii. IEEE Computer Society. August 26 at http://www.informatik.uni-trier.de/ ley/db/conf/hicss/hicss2002-9.html#FlakeM02

[39] Object Management Group. *UML 2.0 Standard Officially Adopted at OMG Technical Meeting in Paris.* Press Release. June 12, 2003. http://www.omg.org/news/releases/pr2003/6-12-032.htm

Appendix A

How to use ChattaBox

A.1 Deployment Instructions

1. Make sure that the Microsoft .NET Framework is installed.
2. Make sure you have a user account on the *machine*. A network authentication is *not* enough due to the required privileges needed by the voice component of ChattaBox (high priority threads) (see Sec. 8.1.1). The user account can be restricted, but belongs preferably to the administrator group.
3. Choose a domain for your Orchestration Server (the main ChattaBox server). Let's suppose that you chose "cs.uct.ac.za".

   Go to /Orchestration Server/Configuration/ and open the file orchestrator.config. Enter your domain name between the <OrchestratorDomain> tags. For example, <OrchestratorDomain>cs.uct.ac.za</OrchestratorDomain>. Save the file.
4. To allow the ChattaBox clients and servers to talk to each other, you need to fiddle with the "hosts" files on the computers that you are going to use. You can find the "hosts" file in /WINDOWS/system32/drivers/etc/, and you need administrator rights on the machine to be able to change it. Make sure there are always a few empty lines at the end of the hosts file.
   - Suppose the computer you want the server to run on is called "netwalker" (and your domain is still "cs.uct.ac.za). You also need to find out the IP address of your computer (netwalker's IP address is 137.158.97.145). Add the following lines to the file and save.

     ```
     127.0.0.1 sip#localhost
     137.158.97.145 sip.cs.uct.ac.za
     137.158.97.145 netwalker.cs.uct.ac.za
     ```

- On all of the client machines, add the following lines to the hosts file.

  137.158.97.145 sip.cs.uct.ac.za 137.158.97.145 netwalker.cs.uct.ac.za

5. Create certificates.
   - Run the Certificate Tool.
   - Go to Wizards, Generate Root Certificate. Save the certificate with default options in /OrchestrationServer/Certificate Store/ as "root.certificate". This will generate two files: "root" and "root.certificate".
   - Go to Wizards, Generate Signed Certificate. For the Root Certificate field, browse for the "root" file. For the Root CA Private Key, browse for the "root.certificate" file. Choose your own Subject Name. The Domain Name must be the same as the domain of your Orchestration Server (e.g. "cs.uct.ac.za").
6. Start the Orchestration Server and create Containers.
   - First, make sure that the permissions on the Debug directory inside the Orchestration Server directory are not read-only.
   - Run the Orchestration Server.
   - Run a Container Server. Suppose you call it "bob".
   - Go back to the Orchestration Server and on the Container Server tab, press the Add Server button. Enter the URL of your Container Server (e.g.tcp://netwalker:5062/bob.server).
   - Add a ChattaBox Container on your Container Server by going to the Containers tab on the Orchestration Server.
7. Now you can run the ChattaBox clients and register users with the Orchestration Server. The server address for the Registration Wizard follows the format "sip".domain_name (e.g. "sip.cs.uct.ac.za").

NOTE: All the user information is stored in an isolated storage. At times things might not work as expected, and you might need to kill the user info in the isolated storage and register as a new user. To do this, go to the Visual Studio .NET command prompt and type "storeadm /roaming /remove".

A.2 Voice Component Configuration

1. The voice component is configured through the "rtp.conf" file in /Client/ executable.
2. The value that most probably needs adjustment is the *SignalEnergy Threshold* used for voice activity detection. Use the utility "VAD Detection Utility.exe". Set it first to 0, the increase until you are only in an talkspurt period when you speak. The utility must be restarted when the value in the file is changed.

No.	Name	Used by; described in section	Suggested Value
1	SignalEnergy Threshold	VoiceActivityDetector; see section 8.1.6	?
2	ConsecutiveSilence Threshold	VoiceActivityDetector; see section8.1.6	20
3	ConcealLoss Threshold	SoundPlayer; see section8.1.5	2
4	OutBufferSize	SoundPlayer; see section8.1.5	4
5	JitterDelay	SoundPlayer; see section8.1.5	3
6	InBufferSize	SoundRecorder;see section8.1.2	20
7	Logging	Logger; see section 8.1.10	0
8	FEC	RtpSender, RtpReceiver. Flag indicating whether Forward Error Correction should be applied or not. Must be the same on both communicating ends. 1 = yes 0 = no [is not implemented with this version of the Voice Component]	0

Table 21: The configuration parameters of the rtp.conf file.

3. Using the same utility, you might also play with the *ConsecutiveSilence Threshold.*
4. The *JitterDelay* should be increased if the voice is often interrupted due to a slow network. Those interruptions qualify by appearing in irregular intervals.
5. The *OutBufferSize* should be adjusted if there are glitches in the playback. On machines with a high performance and a good sound card, it can be tried to be set to a smaller value. This will diminish delay.
6. For all configuration actions, it is a good idea to turn logging on in order to observe the behaviour of the single components. To enable logging, set the *Logging* parameter to 1.

Appendix B

Voice Component API

B.1 Class VoiceComponentControl

B.1.1 Constructors

This class is static.

B.1.2 Public Methods

`startSession(int localPort, char destAddress[], int destPort)`	
Description: This method starts a new voice session.	
Return value: void	
`localPort`	The local port on which the voice component will send audio data
`destAddress`	A FQDN or IP address identifying the peer
`destPort`	The port on which the remote voice component runs

`endSession()`
Description: Ends an existing voice session. The method waits until all threads are
Return value: bool; true if session is successfully ended

B.2 Class SoundRecoder

B.2.1 Constructors

`SoundRecorder(void)`
Description: Constructor for direct access of ADBs through the getADB() method
Return value: void

`SoundRecorder(AudioBuffer* ab)`
Description: Constructor that configures the soundrecorder for use with an AudioBuffer
Return value: void

B.2.2 Public Methods

`start()`
Description: Starts audio recording
Return value: void

`getADB(char* returnADB)`	
Description: Fills `returnADB` with sound data. The start() method must be called befor this method returns sound data	
Return value: void	
`returnADB`	`returnADB[]` must have the size `ADB_SIZE`

`getToAudioBuffer()`
Description: This method first records one ADB and performs voice activity detection on it. If voice is detected, it wraps the data into an RTP packet and puts it into the AudioBuffer. If the packet is the first one in a new talkspurt, the field `RtpPacket.header.m` is set to `1`, otherwise it is set to `0`
Return value: void

`stop()`
Description: Stops audio recording and frees memory allocations
Return value: void

B.3 Class RtpSender

B.3.1 Constructors

`RtpSender(UdpConnection* connection)`
Description: Creates an RtpSender that sends packets to the destination described in `connection`
Return value: void

`RtpSender(UdpConnection* connection, AudioBuffer* buffer)`
Description: Creates an RtpSender that sends packets to the destination described in `connection`, and configures it for communication with an audio source through an AudioBuffer
Return value: void

B.3.2 Public Methods

`send(char* ADB)`	
Description: This method wraps the ADB into an RtpPacket and sends it to the peer.	
Return value: void	
`ADB`	Audio data block. Size = `ADB_SIZE`

`send(RtpPacket* packet);`	
Description: Create update the header for this packet with the current sequence number and send it to the peer.	
Return value: void	
`packet`	an `RtpPacket` containing sound data

`sendFromBuffer(bool wait)`	
Description: Polls RtpPackets from the AudioBuffer and calls send(RtpPacket) to send them.	
Return value: void	
`wait`	if this parameter is set to true, enters a waiting state if the AudioBuffer is empty. This is necessary in the case where the RtpSender runs in its own thread.

B.4 Class RtpReceiver

B.4.1 Constructors

`RtpReceiver(UdpConnection* connection, JitterBuffer* jb)`
Description: This class waits for data to be received from the peer defined by `connection`and puts it into a `jb`
Return value: void

B.4.2 Public Methods

`receiveToJitterBuffer()`
Description: This method waits at most 500ms for the arrival of an RtpPacket with a payload of size ADB_SIZE from a peer defined by a UdpConnection. It then puts the RtpPacket into a JitterBuffer. In the case of a timeout, it returns without doing anything.
Return value: bool; true if the packet was successfully received, false if error or timeout

`cleanupReceiverBuffer()`
Description: cleans up all data in the reception socket defined by `UdpConnection`
Return value: void

B.5 Class SoundPlayer

B.5.1 Constructors

`SoundPlayer()`
Description: Creates a SoundPlayer that is used without a JitterBuffer. Playback is started on the first call of either`play()` or `playSilence()`.
Return value: void

`SoundPlayer(JitterBuffer* jb)`
Description: Creates SoundPlayer that can poll audio data from a JitterBuffer. Playback is started on the first call of one of the three `play()` methods.
Return value: void

B.5.2 Public Methods

`play(char* ABD)`	
Description: Plays the audio data in `ADB`.	
Return value: void	
`ABD`	An ADB of size `ADB_SIZE` containing 8bit/8000Hz/mono PCM data.

`playFromJitterBuffer()`
Description: This method is recommended for use. Can only be used if a JitterBuffer is defined for this class. Polls an `RtpPacket` form the buffer and calls `play()` to play its payload. If the JitterBuffer is empty, it plays either the last successfully played packet or silence. If the packet in the `JitterBuffer` is the first packet in a talkspurt, it initiates a rebuffering of the `JitterBuffer`.
Return value:

`playSilence()`
Description: Plays silence.
Return value: void

B.6 Class JitterBuffer

The underlying `priority_queue` is thread safe.

B.6.1 Constructors

`JitterBuffer()`
Description: Creates an empty JitterBuffer.
Return value: void

B.6.2 Public Methods

`getBufferLoadedEvent(int load)`	
Description: Get a Handel to an event that gets fired as soon as the JitterBuffer has reached a load `load`.	
Return value: `HANDLE`. A handle to an event	
`load`	The load to be reached.

`push(RtpPacket packet)`
Description: Push a packet to the JitterBuffer
Return value: void

`get(RtpPacket* packet)`
Description: Get the packet with the lowest sequence number from the buffer. The packet is copied into `packet` and deleted from the buffer.
Return value: `bool`; false if the JitterBuffer is empty

`size_t getSize()`
Description: Get the size of the JitterBuffer
Return value: `size_t`, the size of the JitterBuffer

`nextSequenceNumber()`
Description: Enquires the sequence number of the next packet. Leaves the packet in the queue.
Return value: `int`

`nextIsNewTalkspurt()`
Description: Enquires whether the next packet is the first in a new talkspurt. Leaves the packet in the queue.
Return value: `bool`. true if new talkspurt

`cleanUp()`
Description: Empties the JitterBuffer
Return value: void

B.7 Class AudioBuffer

This class is thread safe.

B.7.1 Constructors

`AudioBuffer()`
Description: Creates an empty FIFO queue
Return value: void

B.7.2 Public Methods

`push(RtpPacket packet)`
Description: Insert a packet at the end of the AudioBuffer
Return value: void

`pop(RtpPacket* packet)`
Description: Get the packet at the front of the buffer. The packet is copied into `packet` and deleted from the buffer.
Return value: `bool`; false if the AudioBuffer is empty.

`getBufferLoadedEvent()`	
Description: Get a Handel to an event that gets fired as soon as the AudioBuffer contains a packet.	
Return value: `HANDLE`. A handle to an event	
`load`	The load to be reached.

`size_t getSize();`
Description: Get the size of the AudioBuffer
Return value: `size_t`, the size of the AudioBuffer

`isEmpty()`
Description: Enquires whether the buffer is empty or not
Return value: `bool`; true if the buffer is empty

`cleanUp()`
Description: Empties the JitterBuffer
Return value: void

B.8 Class UdpConnection

B.8.1 Constructors

`UdpConnection()`
Description: Creates a UdpConnection container
Return value: void

B.8.2 Public Methods

`create(int myPort, int destPort, char* destAddress)`	
Description: Create a new UdpConnection.	
Return value: `bool`	
`myPort`	The local port on which the voice component will send audio data
`destPort`	The port on which the remote voice component runs
`destAddress`	A FQDN or IP address identifying the peer machine

The following methods are self-explanatory.

```
sockaddr_in getLocalInfo()
sockaddr_in getRemoteInfo()
int getSocketID()
void makeSocketBlocking()
void makeSocketNonblocking()
```

B.9 Class VoiceActivityDetector

B.9.1 Constructors

`VoiceActivityDetector()`
Description: Create a new `VoiceActivityDetector`
Return value: void

B.9.2 Public Methods

`voiceActivityDetection(char* audioData)`
Description: Decides whether this `audioData` contains voice data or silence
Return value: `bool`; true if voice is detected.

`getLastSpectralEnergy()`
Description: Get the spectral energy of the latest analyzed packet
Return value: `double`; the spectral energy

B.10 Class Properties

B.10.1 Constructors

`Properties()`
Description: Reads the the first line of the file `rtp.conf` in the same directory and extracts all property values.
Return value: void

B.10.2 Public Methods

```
int getInBufferSize()
int getOutBufferSize()
int getJitterDelay()
int getConsecutiveSilenceThreshold()
double getSpectralEnergyThreshold()
int getConcealLossThreshold()
int getDoLogging()
int getFECflag()
```