A mobile SIP Client

From the user interface design to
evaluation of synchronised playout from multiple SIP user agents

ATHANASIOS KARAPANTELAKIS

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2007

COS/CCS 2007-07

A MOBILE SIP CLIENT: FROM THE USER INTERFACE DESIGN TO EVALUATION OF SYNCHRONISED PLAYOUT FROM MULTIPLE SIP USER AGENTS

by

Athanasios Karapantelakis

Submitted in partial fulfillment of
the requirements for the degree of

Master of science (Information Technology)

at the

School of Information and Communication Technology
Royal Institute of Technology
Stockholm, Sweden

Advisor and examiner: Professor Gerald Q. Maguire Jr.

27 February, 2007

Abstract

The thesis examines the ability to have synchronized playout of audio from multiple devices. The paths between the source of this audio and the output devices may be different. Our objective is to provide a generic low-cost solution, without the use of special hardware.

The context of this work is internet telephony, but the work is also applicable to other settings. In internet telephony this synchronization not only contributes to the natural flow of conversation, but also maintains the spatial effect of the incoming audio streams, as well as the location awareness of the peers.

We envisioned users of handheld devices might collectively utilize their personal devices to enable a shared spatial audio environment. In the simplest case two users each with monophonic terminals could provide stereo. Hence, the second part of this study addresses the practical issue of how such synchronization could be utilized in a internet telephony client to provide such multidevice playout. We utilized minisip, as an open-source Session Initiation Protocol (SIP) client supporting security, as the basic client. To realize the vision, we ported minisip to a Pocket PC environment. In the second part of this thesis we examine the process of porting preexisting code to such a new architecture, as well as how to map an existing human-computer interface to such a handheld device.

The thesis shows that synchronization is possible and explores some of the implementation's limitations. A clear result is the need to combine the results of several theses into the common trunk code - so as to combine the earlier RTCP work with this synchronization work and to provide the a human-computer interface which is suitable for many different handheld devices, ranging from Pocket PCs to SmartPhones.

Sammanfattning

Rapporten visar på möjligheten att synkronisera ljuduppspelning på multipla ljudenheter. Vägarna från ljudkllan till de olika högtalarna (utenheterna) kan skilja sig. Vårt mål är att tillhandahålla en generell lösning till en lågt kostnad, utan att behöva använda specialhårdvara.

Området för detta arbete är internettelefoni, men arbetet är även tillämpbart inom andra områden. I fallet med internettelefoni så bidrar ljudsynkroniseringen inte enbart till det naturliga konversationsflödet, utan även till de rumsrelaterade aspekterna av de inkommande ljudströmmarna och samtalsparternas medvetenhet om sina geografiska positioner.

Vi förutser att användare av mobila terminaler kan komma att använda sina terminaler tillsammans för att möjliggöra en gemensam ljudmiljö. I sitt enklaste utförande kan två mono-enheter tillsammans skapa en ljudmiljö för stereo-ljud. Därför adresserar den andra delen av studien hur denna typ av ljudsynkronisering kan användas inom IP-telefoni för att möjliggöra synkroniserad uppspelning på flera enheter. Vi använde minisip, en klient för SIP byggd på öppen källkod och med säkerhetsstöd, som en grundläggande terminal. För att realisera vår vision så portade vi minisip till Pocket PC-miljön. I den andra delen av den här rapporten undersöker vi även processen för att portera existerande kod till en sådan arktitektur, och hur man överför existerande användargränssnitt till en handhållen terminal.

Denna rapport visar att synkronisering är möjlig men visar samtidigt på en del av begränsningarna i implementationen. Ett tydligt resultat är behovet av att kombinera tidigare rapporters resultat – för att kombinera tidigare arbete inom RTCP med detta arbete inom synkronisering och för att tillhandahålla ett användargränssnitt lämpat för många olika handhållna terminaler, från Pocket PC-baserade till SmartPhone-baserade system.

Contents

Chapter 1

Introduction

This thesis examines the question of synchronized audio output from multiple devices which may have different networks paths from the source to the device. One of the purposes of such synchronized audio output is for spatial audio (see section 1.2). The usual means of providing synchronized playout uses a single device with multiple audio outputs. However, the case we consider here is the user of multiple network devices each with at least one audio output. For example, it could be two friends who each have a device with a single speaker and would like to listen to a remote concert in stereo, a group of friends who want a sound-sound experience, or for a user participating in a teleconference to virtually place the other speakers around a virtual table. This chapter introduces the basic concepts used in the remainder of the thesis and gives a statement of the specific problem to be solved (see section 1.3.2).

1.1 General principles

1.1.1 The nature of sound

Acoustics, the branch of physics that studies sonic related phenomena, defines sound as mechanical waves, generated by fluctuations in mechanical energy, and subsequently propagated in space, through a medium such as air, water, or earth. Propagation is achieved through the compression of particles, deviating from their original position. Sound waves are detected by the human auditory system, when the eardrum of the ear vibrates, as an after effect of the particle compression.

Sound waves inherit the standard characteristics of any waveform. Frequency and amplitude are two of the fundamental characteristics of waves, both playing an important role on how humans sense sound. The audible spectrum of frequency for humans is between 20 Hz and 20 KHz, although it is known to shrink with old age. As a rule of thumb, the higher the frequency of the sound waves, the higher the sound pitch.

The amplitude of a wave, is a measurement of the magnitude of oscillation of the wave. In other words, amplitude measures the degree of displacement of the particles in the medium the wave traverses. Amplitude is directly related to sound intensity, thus the greater the amplitude, the greater the sound volume. However, the perception of volume is related to the log of the intensity.

There are also a number of other properties to sound waves, some of them related to their waveform nature, and others to environmental interference [1].

1.1.2 The human auditory system

The first step towards constructing a model for spatial audio, is to comprehend the fundamentals of the human auditory system. Our sense of hearing relies on two main input sources, namely the right and the left ear. Additional input sources include the skin and bones; however, we will not include this in our analysis here since we will largely restrict ourselves to frequencies greater than that which we feel as vibrations with our bodies and also at lower sound volumes than that which strongly shakes the bones. For simplicity we will also ignore bone conduction of sound to the ear. The human auditory system is not only capable of processing sounds in the audible spectrum, but also recognizing the apparent spatial orientation of sound. Psychoacoustics is the branch of psychophysics[1], that concentrates on the human perception of sound. Historically, Lord Rayleigh (1900), was the first to propose a simple model for binaural [2] sound localization. Lord Rayleigh's proposal was based around a rudimentary circular-shaped human head and is known as the *duplex theory of localization*. According to this theory, the sense of direction to the sound source, is based on two different interaural attributes, otherwise known as binaural localization cues.

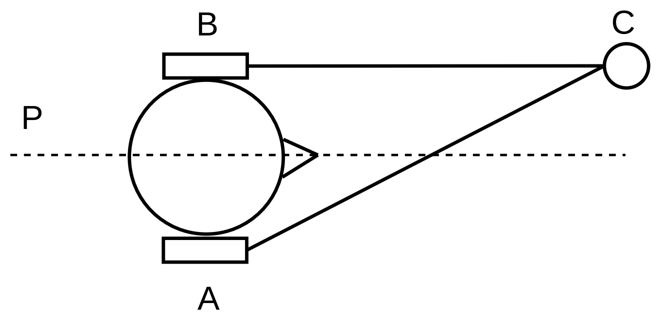


Figure 1.1: Median plane P is defined as the intersection of the head, bearing equal distance from sound detectors (ears) A and B. P divides the space into two subplanes. Sound detector B and sound source C belong to the same subplane, so the sound wave reaches ear B first.

[1]Psychophysics is a subdiscipline of psychology.

[2]Binaural (as used in this document): having or relating to two ears.

The first cue highlights the time difference of the sound wave reaching the two ears. In most cases, the sound wave has to traverse a greater distance to reach one of the two ears (figure 1.1). This temporal disparity in arrival time is called Interaural Time Difference (ITD). The maximum ITD that humans can perceive is 630 microseconds [2]. Because of this short time frame, ITD is useful for low frequency sounds [3], such as those with frequency within the audible spectrum (see section 1.1).

The second cue, *Interaural Level Difference* (ILD), concerns a reduction in signal intensity, for one of the ears. The phenomenon occurs due to the obstruction of the sound path, to the ear further from the sound source (figure 1.2). In particular, the sound wave reaching the far ear will be attenuated, as it will be blocked by the head. As is the case with ITD, ILD does not perform uniformly across the frequency spectrum. For frequencies under 1.5 KHz, sound waves diffract around the head, since the wavelengths are longer than the head's diameter. Above 1.5 KHz, the wavelengths are shorter than the diameter of the head, thus the sound waves traverse the head. This process attenuates the signal [4].

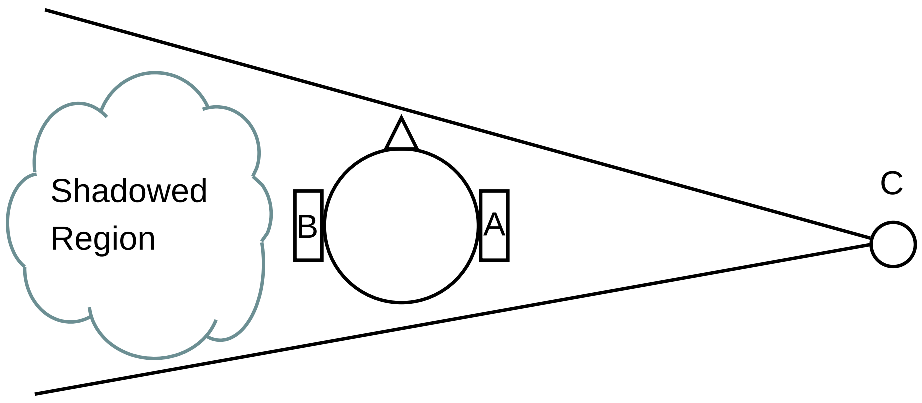


Figure 1.2: Head obstructs sound waves from sound source C reaching ear B directly.

1.2 Sound localization

Spatial audio is a term related to sound that has been processed in order to give the listener a sense of space, thus emulating natural sound sources a person can hear, in a virtual acoustic environment. Spatial audio can be useful in situations where a listener would rely on sound to establish the origin of a source, or would be in a multi-aural environment, enabling the listener to distinguish one source from another. The application field for spatial audio is quite broad, and among others includes the following categories:

- Applications where the conversation participants are not visible to one another

An example is a spatial-audio system developed to make use of the Global Positioning System, so that users are able to infer each other's position, (near, far, which direction), just by using their sense of hearing. Holland et al. [5], argue that there are situations where users might not want to be distracted with having to use a visual user interface in order to interact with other people (for example an instant messaging service), or are in such a highly mobile and dynamic environment, that they simply do not have the time to use such interfaces. Their pilot implementation takes advantage of the GPS system to determine the relative position of each user, using direction and distance as metrics. Then the sound is modeled according to these metrics and is presented to the listeners of the conversation.

- Applications in virtual environments

  Virtual environments, especially those that feature three dimensions, can benefit a lot from spatial audio, conveying the three-dimensional sense more convincingly to the end user. Such environments are often utilized in games. The gaming industry has experienced increasing revenues over the last decade, and could benefit significantly from spatial audio research. Spatial audio can take gaming experience to the next level, by providing more realistic and natural sounds. Drewes et al. [6] have implemented a game that features an immersive audio and visual experience.

  Another area of interest is VRML (Virtual Reality Modeling Language). The VRML standard [7] specifies a file format for representing three-dimensional scenes. Computer Aided Design professionals can also benefit from this technology. An example is modeling the engine noise from a car's engine, which penetrates into the interior of the vehicle.

- Warning systems

  This category includes a broad range of applications, from civil and military aircraft warning systems, to personal paging systems. Naturally, the scale and the critical importance of these systems vary greatly. Shilling et al. [8], evaluate a spatial audio interface designed for military use. Specifically, the system works in correlation with a display, and warns helicopter pilots about threats. It also assists in issues of navigation and targeting.

Other applications include use of spatial audio in large conference rooms, auditoriums and cinemas. Also, last but not least, telephony applications. Spatial audio in telecommunications is of great importance since it can support a number of potential categories of interest, combined with the flexibility of the wide area coverage of both fixed and mobile telecommunications networks. This idea can further be extended for use in Internet telephony [9]. One of the fundamental changes which spatialized audio introduces in internet telephony, is the ability for the user to listen to more than one call at a time - which fundamentally changes the usage model for telephony.

1.2.1 Issues with Spatial audio

- Cost.

  Historically, the first spatial audio systems were used during World War 2, but their cost and size rendered them infeasible for private users. During the 1990s the first analog devices started to emerge [10], that started to make more sense to the consumer market, both in terms of price, and in terms of volume. Today, digital signal processors (DSPs) are routinely used to produce spatial audio. A consumer can purchase DSPs for his/her computer, or built into the computer at a fraction of the cost of these early systems.[11].

- Evaluation

  Up until recently there has been no general consensus concerning how to evaluate spatial audio systems. Although metrics such as synchronization delay, echoes, noise, etc., can play a role in the evaluation procedure, it is unclear what other factors are significant. Furthermore, the impression of a 3D audio experience may vary from person to person, so the human factor is also very important in such evaluations. Rumsey [12] discusses the issue of spatial audio quality, why it is important, and suggests ways to evaluate in. Rumsey argues that according to the objective of each system, different levels of detail must be employed, and that objective measurement models must take into account motion in some audio scenes, thus requiring fluctuations in the quality of the audio signal. Currently, the majority of the evaluations have focused on static spatial audio environments.

- Software

  Although spatial audio software may use signal processing hardware (such as the widespread personal computer sound hardware), it usually lacks support for what some may regard as critical issues, such as environmental modeling, synchronization, and network support [13]. Another issue has been the effort and the knowledge required to construct spatial audio software. This was due to the fact that complex signal analysis calculations had to be implemented in a programming language, such as C. In order to interface with the various DSP boards, the assistance of the special libraries from the DSP board vendor were needed, making the software more complex, and *vendor-specific.* Recently, software packages have emerged that allow programming of DSP processors, without requiring prior knowledge of C or the details of the specific DSP board. Instead, they provide a consistent user interface and support a number of different DSP boards. Examples include Alladin Interactive DSP[14], and LabView DSP[15]. In addition, hardware devices known as DSP emulators, allow the development of software on different DSP architectures using a high level symbolic language [16].

- Environmental Modeling

  Spatial audio greatly depends on the surrounding environment. In the physical world, phenomena such as echoes and reverberations occur in larger or smaller magnitude, depending on the physical structure of the environment, which must be replicated in the virtual environment. The problem of modeling the 3D environment, is quite complex, and there are multiple solutions. Kurniawan et al. [17], classify the methods as numerical, geometrical, and statistical. Most spatial sound systems employ the statistical method, which uses statistical models for echo and reverberation. In simple terms, although this method is not correct, it can create realistic results. In the settings which this thesis will explore, we assume that some source has generated the correct audio streams (using which ever method they have chosen), and we will focus on *rendering* (i.e. playout) of these streams.

1.3 Context of this study

1.3.1 Streaming spatialized audio

Traditionally, spatial audio systems were intended for local audiences and participants. However, over the last decade, in par with the popularity of the internet and tele-conferencing applications, many virtual communication environments with localized sound have started to appear. Examples include remote conferencing applications [18], air-traffic control [19], television broadcasts [20] etc. Such systems, are not bounded by the limitation of local presence of participants and audience, since audio content is delivered to the peers by taking advantage of the underlying network infrastructure.

Stereophonic sound is the simplest form of spatial audio, setting it apart from monophonic recordings. There are two voice channels, conceptually named ”left” and ”right”. More advanced systems utilize six, eight, or more audio channels. As is often the case with virtual spatial audio systems, streaming multiple channels over heterogeneous networks, can result in asynchronous delivery of the audio content, due to different network delays for different streams.

1.3.2 Problem statement

This thesis implements a software framework to synchronously deliver spatialized voice streams to multiple recipients, over network links with different delays (this could be caused by use of different network paths or congestion). Network congestion can be the cause for a series of phenomena that affect the voice quality due to both packet loss and varying packet delivery delay.

This work is not related to the study of a single audio stream, but rather efficient synchronization of multiple streams. As it will be discussed in chapter 3, synchronization is not only beneficial for conducting natural conversation between multiple repicipents, but also for sound spatialization environments. A specific case study involves two users, one who records and sends a stereo voice stream (U1) while the other (U2) receives and listens to the recording. Figure 1.3 illustrates the scenario. Assuming that device S1 sends a stereo audio stream of U1's voice and devices D1 and D2 are responsible for playing the left and the right channel of the voice stream respectively, is it possible for U2 to listen to the stereo stream exactly as it was transmitted? Maintaining the stereo effect during the playout on D1 and D2 is not a sraightforward issue, considering the different delays the packets may experience on the network links (S1, D1) and (S1, D2).

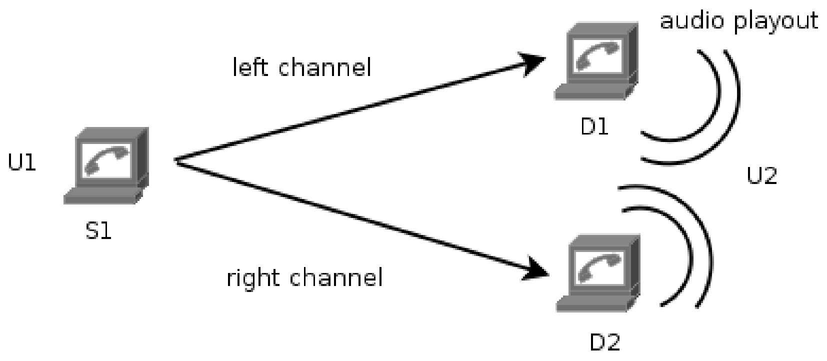


Figure 1.3: Simple scenario of voice stream synchronization.

1.3.3 Issues concerning voice transmission over congested networks

- Jitter

  At the receiver, jitter can be perceived as abnormal variations in the voice packet interarrival time. The greater the deviation from the stable packetization time on the sender, the more the jitter. Excessive jitter results in choppy playout (figure 1.4). Mathematically, jitter can be represented as the standard deviation from the voice packet inter-transmission time. The simplest way to address this problem is to add a buffer, that stores the incoming packets for longer than the greatest expected jitter which is to be accepted, thus the audio samples can be read in a timely manner. Such buffer will also introduce additional end-to-end delay. Research is being done for adaptive buffering, that makes adjustments according to current network conditions [21] (see section 6.1). Such adaptive buffering assumes that the delay variance (jitter) can be estimated and tracked. As we will see later in section 3.1, the first of these assumptions places limits on how much data we need to make such estimates and the second assumption bounds the maximum rate of change in jitter to which we can adapt.

- End-to-end delay

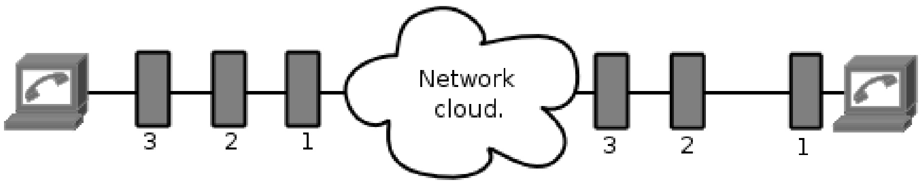


Figure 1.4: Variation of packet interarrival time, also known as jitter. A network cloud, as used in this study represents a network path with an arbitrary number of network devices such as routers and switches between its entry and exit points.

Intermediate network components in the transmission path, such as switches and routers may add additional delay. Stable delay does not affect voice quality until the end-to-end delay exceeds some limit for interactive conversation (since the packet interarrival time does not deviate from the packetization time), but it may result in an unnatural and inconvenient conversation. A common example is that of a news correspondent reporting via a satellite link for a news broadcast. There are usually pauses between the questions from the anchorman in the studio, and the correspondent's answer. These silence gaps are generally attributed to the delay added by the communication sattelite relay network.

- Packet loss

  As the name suggests, packet loss occurs when packets which are sent do not reach the receiver. Constant packet loss has a negative effect in voice quality (Figure 1.5). However, a lot of research has been conducted concerning both forward error correction and Packet Loss Concealment (PLC) algorithms, the first method adds additional redundancy to the streams to mitigate the effects of packet loss and the second attempts to mask the effects of packet loss on voice streams. The choice of a suitable algorithm is a tradeoff between quality and simplicity. Simpler algorithms tend to copy previous samples, or following simple repetition models, whereas complicated ones are focused on modelling the speech. Simplicity is an important factor, especially in environments with limited resources, for example an ad hoc network of handheld devices. Small percentages of packet loss rate, can be addresed by simple algorithms, whereas large loss rates may result in incomprehensible speech, without an efficient modelling scheme [22].

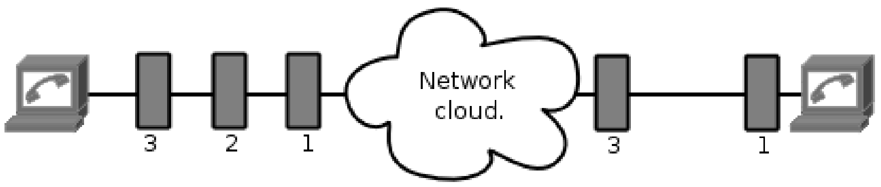


Figure 1.5: Network congestion or link failure can result in packet loss, a situation in which packets are lost on the transmission path.

- Sequence errors

  In the presence of multiple routes between the sender and the receiver, congestion can cause packets to be routed over different network paths and packets can be processed in various orders due to queuing and other processing in routers, thus arriving at the receiver out of order. If the receiver attempts to play the audio data contained in the packets using a strict First In-First Out (FIFO) policy, the result will be garbled sound (Figure 1.6). However, using the sequence numbers it is possible to insert the samples into their correct temporal order - provided that there is sufficient buffering so that the samples have not arrived too late.

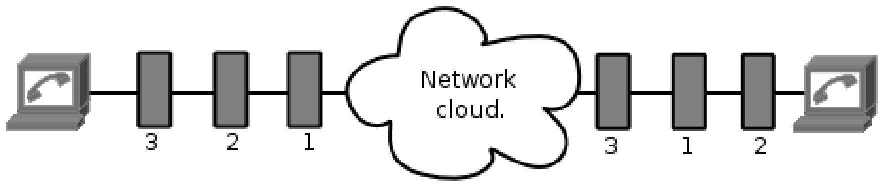


Figure 1.6: Sequence numbers help the receiver identify the correct order of packets. Media protocols such as Real-time Transport Protocol (RTP), include sequence numbers in their headers.

Chapter 2

A framework for synchronous playout of spatialized audio

2.1 Architecture overview

The block diagram in figure 2.1, illustrates the discrete components of the system. The subsequent sections of this chapter describe these components, from the underlying communication and synchronization protocols to the software applications and hardware infrastructure. The final section presents some real-world examples, where this architecture might be applicable.

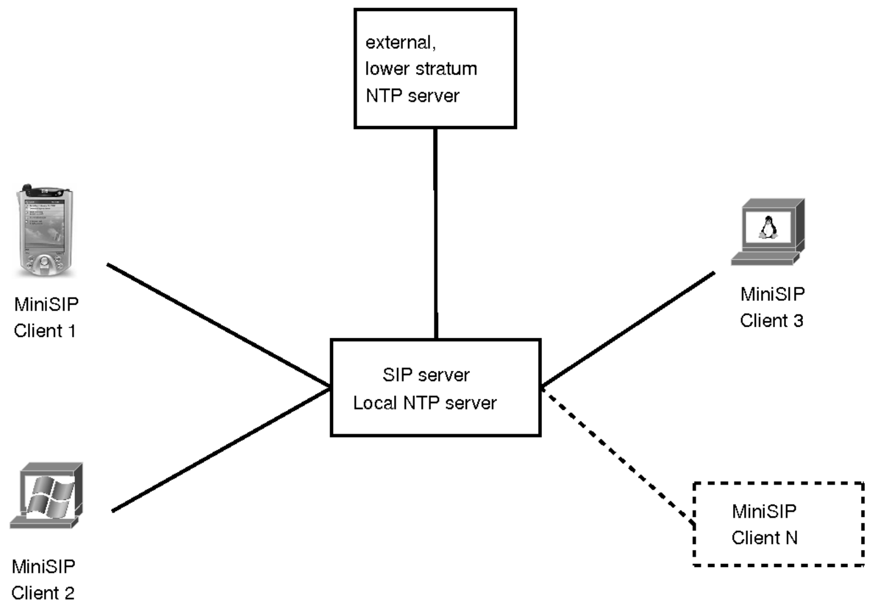


Figure 2.1: Components of the system. Users can participate in the conversation with a variety of devices, such as PDAs, GNU/Linux or Microsoft Windows computers (section 2.4). Each device synchronizes its local clock with the same time server (section 2.5.2). When in conversation, timing information from the local clocks of the clients are combined to measure the delay of the audio channels.

2.2 Session establishment and transport protocols for voice calls.

The Internet Protocol (IP) [23], is the predominant network layer protocol used by applications and networks today, including the Internet. On top of IP, the connection-oriented Transmission Control Protocol (TCP) [24], and the connectionless User Datagram Protocol (UDP) [25], reside in the transport layer. The usage of UDP/IP or TCP/IP protocol stack depends on the nature of the network service. For example, a teleconferencing application is likely to use UDP/IP, in order to avoid the jitter and unnecessary delays caused by TCP retransmissions (see section 1.3.3). On the other hand, an application that requires reliable data delivery, such as a credit card reader and charging client, would want to use TCP/IP, to ensure reliable delivery of transactions. In some cases, the responsibility for delivery reliability can be delegated to the application layer. Naturally, in the case of the UDP/IP stack, there is no alternative.

The Session Initiation Protocol (SIP)[26], builds on top of the TCP/IP and UDP/IP stack, offering a way to create and terminate media sessions[1] involving two or more participants. Although there are other session protocols offering similar functionality, they either lack in scalability and interoperability [27] , or they are unavailable in the public domain [28]. Adoption of SIP from major software and hardware vendors (such as Cisco, Microsoft, and Motorola), acknowledges SIP as one of the dominant protocols for media call control, at least in the near-term future. In addition, SIP is considered to be an easy protocol to implement and debug, due to its simplicity, and clarity of the draft specification. For the reasons stated above, this study uses the SIP protocol for control of peer calls.

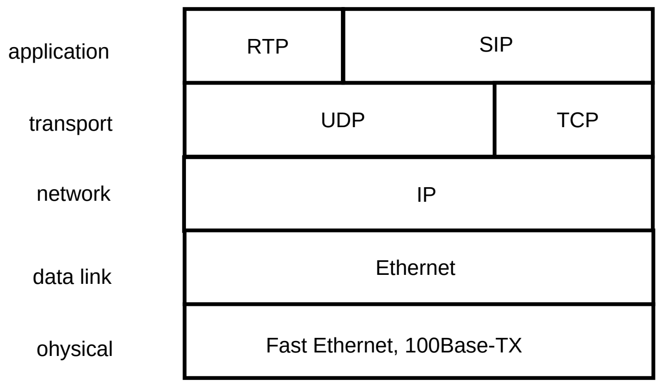


Figure 2.2: The protocol stack used in this study, with layer description on the left.

The majority of implementations complement SIP, with the powerful and flexible Real-time Transport Protocol (RTP) [29]. This protocol is responsible for transporting and delivering the audio data[2] to peers during SIP sessions. It is worth mentioning that the RTP draft does not

[1]Media sessions refers to audio or video sessions, or both concurrently. Although this thesis focuses on audio transmissions, SIP is capable of establishing video calls as well.

[2]Again, RTP is capable of transporting audio, compressed video, timed text, and other data, but our references

prohibit connection-oriented protocols (i.e. TCP) to be used as the underlying transport layer. However, as stated above, using such protocols may result in unwanted jitter. In our experiments we make use of the RTP/UDP/IP stack (figure 2.2). In addition, while there is no direct solution to the issue of timely and orderly arrival of RTP voice packets[3](see section 1.3.3 for sequencing errors), each RTP packet carries a sequence number, helping the application to detect packet losses, and reorder the packets in correct order, before playing the audio stream. The entire stack mentioned above, provides a framework for voice communication over IP networks, otherwise known as Voice over IP (VoIP).

It is also important to clarify that even if the SIP messages that initiate the session between the peers are routed through the SIP proxy, the RTP traffic is not (figure 2.3). Instead, once the IP address of the peer becomes known through the SIP message exchange with the proxy, RTP packets are routed directly between the peers, just as any other network packet. More formally, the IP and port numbers that are of concern is the IP address and port numbers to be used by the RTP audio stream(s). Note that these may differ rom the IP address of the client, see for example the thesis of Oscar Santillana [31].Therefore, the proxy server in this architecture has no other role than to provide global timing (see section 2.5.2) and peer IP address information when prompted. There are many open source SIP proxies available. One of the simplest and more robust ISIP proxies is SIP Express Router (SER) [32], which was used in our experiments (see chapter 4).

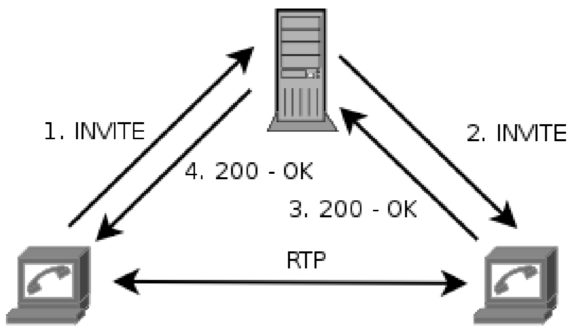


Figure 2.3: After an initial SIP message exchange through the SIP proxy (messages 1 to 4 above), the peers learn about the IP addresses of each other. The actual voice stream of RTP packets, is routed directly between the peers, just like any other IP traffic.

limit to audio in this study.

[3]Disambiguation of RTP packet and frames: Throughout the literature, the terms *packet* and *frame* are used interchangeably. In RTP, frames make sense when used together with frame-based audio and video CODECs. In this case, it is possible to encode several consecutive frames within a single packet [30]. Frames are regarded as pure audio or video data, and it is up to the codec to provide guidelines for the size of each frames, and the number of frames that can be included in one packet. An RTP packet in turn, has the RTP header in front, accompanied by one or more frames as the payload. This thesis uses the G.711 voice codec (see section 2.3), which is sample-based. What this means is that the packet size is determined from the audio sampling rate. By implication, the term *RTP frame* is not relevant to this study, and will not be used henceforth.

2.3 Audio Codec

An audio codec is responsible for converting data to a certain format. Audio data are encoded in this format prior to their transmission, then decoded at the receiver. The codec used in this study is G.711. The specifications for G.711 are published by the International Telecommunication Union (ITU). The sampling rate is set to 8,000 samples per second. In addition, when using this codec with RTP packets, it is recommended that the typical packetization interval should be 20ms [30]. Using this data, we can calculate the payload of G.711 encoded data for each audio packet:

$$intervals/sec = \frac{1000msec}{20msec} = 50$$

$$packetpayload = \frac{8000samples/sec}{50} = 160samples$$

The conclusion is that an RTP packet is transmitted every 20ms, a duration which is also known as packetization time. Furthermore, in order to be able to match the sampling rate of the audio data and avoid losses, each packet must have a payload of 160 samples. As it is discussed in chapter 3, this information is critical to the success of the synchronization mechanism. It should be further noted that these samples can be monaural or stereo and each sample is encoded to an eight bit value.

2.4 SIP Client

A SIP client is a piece of software running in every peer's terminal during a VoIP session. Usually it associates with a SIP registar (a type of SIP proxy), a central database that serves as a repository for resolving peer addresses. Minisip [33] is a SIP client, developed at Royal Institute of Technology, in Stockholm, Sweden. Minisip is the result of the collaboration of many people, and has a unique set of features, making it an ideal choice for the purposes of this study (in our case two features are of particular importance: spatial audio and security of both the media stream and the call setup).

One of the fundamental requirements, was cross-platform compatibility, so that our framework can operate on diverse hardware architectures and operating systems. Minisip already runs on both Windows and GNU/Linux operating systems (specifically the debian derivative distributions and Fedora core), on Intel's x86 architecture. In addition, minisip is free software[4]. This thesis project also contributed to the minisip effort, by attempting to port minisip to the Pocket PC environment, under Windows CE operating system, using Microsoft Corporation's compilers for the ARM platform. The port process involves the implementation of a graphical user interface (GUI),

[4]The free software foundation disambiguates the terms free and open source software. [34]

providing similar functionality to the preexisting GUI for Windows and GNU/Linux. Chapter 5 describes the porting process in detail.

Previous work on the client [9], has also made it possible to support wideband resampling, from the original 8Ksamples/sec, to 48Ksamples/sec, leading to better and more illustrious voice quality and the ability to mix this audio with entertainment audio. Of particular interest to this project is the audio spatializer that works in conjunction with the resampler. After the incoming voice streams are resampled, the spatializer uses an algorithm to localise them in virtual positions around the listener. Although these positions in the spatializer are currently hardcoded, section 3.3 presents a model for peer location awareness, that automates this process.

Another effort added support for the Real-time Transport Control Protocol (RTCP) to minisip. RTCP provides feedback on the quality of RTP streams [35]. RTCP-enabled peers periodically transmit control packets over the same network path they are exchanging RTP audio data. Although RTCP packets do not carry any media themselves, they provide information about the quality of the audio data (jitter, packet loss, and round trip delay). Of particular use to our framework, is that they can be used to provide information about the one-way voice channel delay, aiding in stream synchronization (section 3.2).

2.5 Synchronizing incoming voice streams

2.5.1 An algorithmic approach to synchronization

With regard to the scenario illustrated in figure 1.3, the question is how to synchronise two streams traversing through different network paths, which may have quite different one-way delays. For reasons of simplicity, we study the network stream from the sender to the receiver, as this is the path that is relevant to the processing of the RTP packets which are being set to these destinations. Since the receiver and sender are roles, these roles reverse in the obvious way in a natural two-way conversation. However, the underying process involves two one-way streams in opposite directions, hence the proposed algorithm applies in the two-way conversational case as well. While synchronization at first appears to be a simple problem, it is in fact comprised of a number of smaller issues that need to be examined separately:
a.) First we need a means to estimate the one-way delay from the source to the playout device for both of the network paths (here we are assuming the simple case of two devices, but the problem generalizes to more than two devices).
b.) Next we must compare the delay of the network paths and extract the difference in delays. This difference will be referred to as the relative delay from now on. It can be written symbolically

as:

$$d_{rel} = |d_{linkA} - d_{linkB}|$$

Where d$_{rel}$ is the relative delay, d$_{linkA}$ and d$_{linkB}$ are the one-way delay on the first and second network links respectively.
c.) Finally, the link that has the lowest network delay must wait for a period of time at least equal to the relative delay before playing the incoming audio data. For this reason, the incoming data on this link, should be temporarily stored in a buffer long enough to support the needed waiting time. After this time has elapsed, both clients on both network links can start playing the audio data which they have received.

The rest of this chapter discusses the technical approaches taken to address each of the steps described above.

2.5.2 Using NTP data for delay estimation

The Network Time Protocol (NTP) is used to synchronise clocks of devices with a global time source (also known as reference time source). Reference time sources range from high precision atomic clocks, to low cost quartz oscillators embedded into the motherboards of modern personal computers. In order for a device to synchronise its clock against an NTP server, a functional network path to reach that server is required. The NTP draft specifies a hierarchical structure for NTP servers. Each layer of the hierarchy is called stratum. A server of stratum n (where $n \geq 2$), provides reference time to timeservers of stratum n + 1 and to clients which place requests. In addition, it synchronizes its clock with the reference time provided from servers of stratum n - 1. There are special regulations for servers of strata 1 and 0. Stratum 0 is a set of high precision devices, such as atomic clocks, or GPS receivers getting time from sattelites. These devices are only accessible from stratum 1 servers. Each stratum 1 server shares a direct connection with one of these devices.

NTP time synchronization can be used to help find the one-way delay on a network link. Figure 2.4 illustrates a network path between two clients. For reasons of simplicity, we assume that the traffic is one way, from client A to client B. Both clients A and B keep their clocks synchorinized with a local NTP server. In our architecture (figure 2.1), the reference time is provided by the local SIP registar, which for the purposes of our tests assumes the role of a local NTP server as well. This server in turn, synchronises its clock with a lower stratum NTP server. Assuming that the clocks of A and B are synchronized, a packet marked with A's timestamp on departure, can be compared with B's clock time upon arrival. The timestamp difference directly yields the channel delay from A to B:

$$d_{linkA} = |t_B - t_A|$$

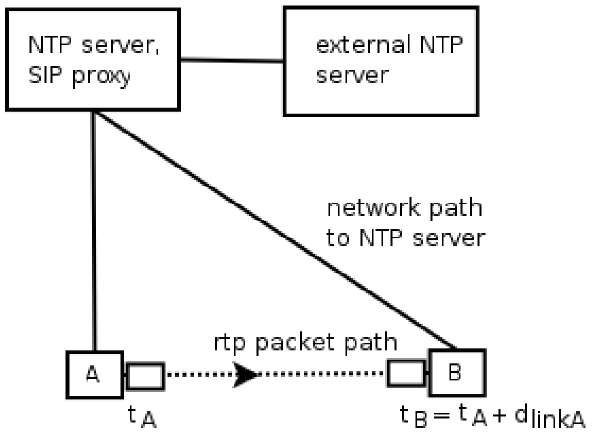


Figure 2.4: Calculating one-way delay on the network path from A to B using syncd packets.

However, this NTP based mechanism requires another network protocol, in order to deliver timestamp information to the peers. RTCP (section 2.2) is a protocol suitable to address this requirement. RTCP packets are small in size, so they do not consume too much bandwidth and they are transmitted periodically, allowing a continuous monitoring of the channel delay. Furthermore, they include a timestamp of the sender's clock in their header. This timestamp can be compared against the local clock, as stated above. The reader should note that the purpose of this protocol is simply to relate the sender's RTP time stamp to the sender's NTP synchronized clock. This is necessary because the RTP timestamp is initialized to a random value at the start of the RTP stream and simply increased based upon the sampling clock of the stream. Thus for our case each tick of the RTP time stamp will represent 1/8000 of a second.

Unfortunately, RTCP is currently only implemented in minisip for the Windows and Windows CE operating systems. In addition, Windows has faced long-term problems with precise timekeeping. In light of these shortcomings, a small library called "syncd" was developed. Its sole purpose is to send and receive UDP packets with the sender's local timestamp as payload, this is sufficient to provide both the relationship between the sender's NTP synchronized clock and the RTP timestamps, and it also provides as a secondary benefit an additional measurement of the network path's current delay. Section 3.2 provides an overview of the implementation and compares syncd with the RTCP alternative.

2.6 Case studies

2.6.1 Scenario A: Casual conversation

Figure 2.5 illustrates a potential scenario. Dave is participating from an uncongested network from New York, whereas Bob connects via satellite phone, from the Nevada desert with scarce bandwidth. Alice should be able to hear Bob, Carol, and the others as if they were in the same room. Bob is to be virtually located far to the left, and Eve is near to the right. (note the relativity of terms far and near). In order for the conversation to sound smooth and natural, the system should be able to apply the knowledge of the spatialized audio model, in order to determine a corresponding relative delay to the streams Alice receives. Also, the system has to maintain this relative stream delay regardless of the network conditions[5]

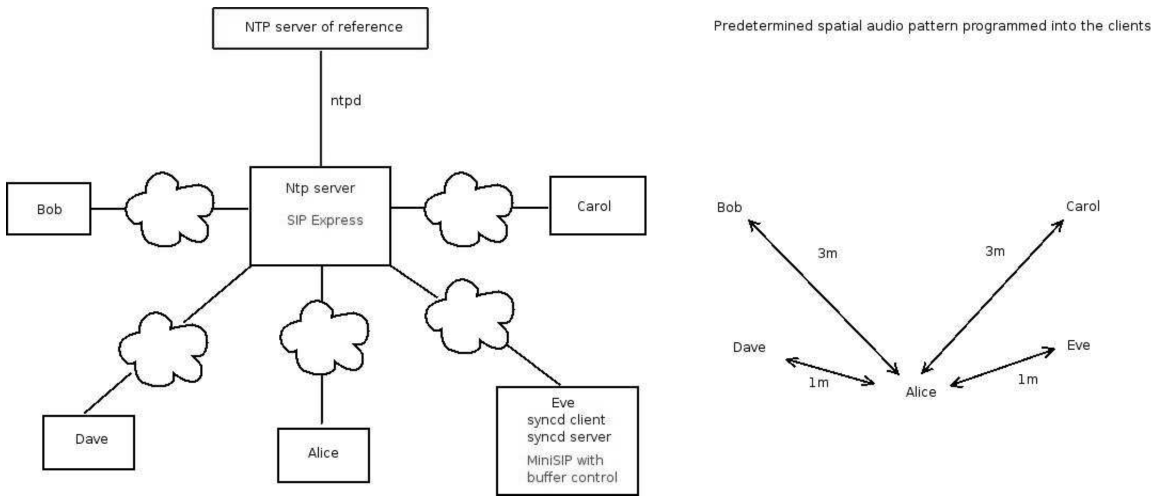


Figure 2.5: A session with participants from around the world.

2.6.2 Scenario B: Conversation of critical importance

This scenario describes a situation where stream synchronization is important. An airport control tower operator is in the process of assigning precedence and giving landing clearance to two aeroplanes. Both pilots in the two planes must be able to hear the same command at the same time, regardless of their distance from the control tower, and the weather conditions that may affect the signal (figure 2.6). In a simplified example, the control tower authorises the Boeing plane to land, since there currently is only space for one plane to land at a time. While Cessna acknowledges that order, this is not heard in time by the Boeing, which will continue approaching

[5]In this case, the spatial model is known from the beginning, so it is preprogrammed into the minisip client of each participant. Also note that the spatial-aware model described in section 3.3 would not be applicable in this case, since it is based in capturing soundwaves from other peers, requiring the peers to be close proximity so they can hear each other.

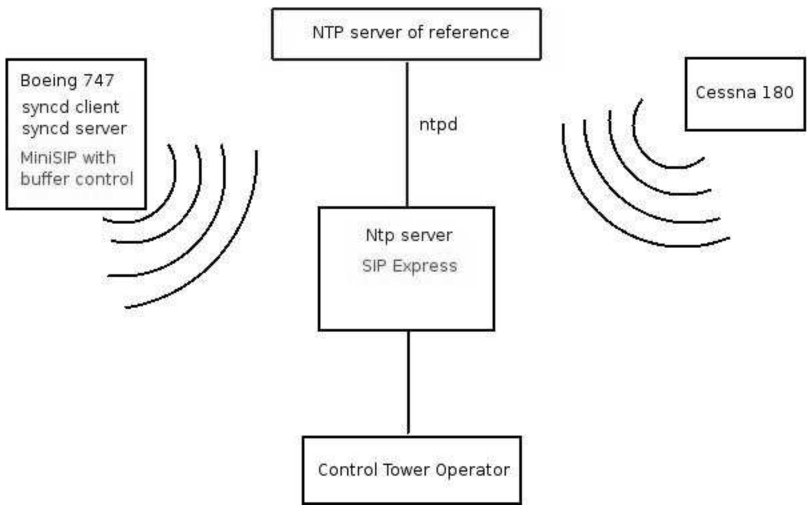


Figure 2.6: Synchronous coordination of approaching aircrafts.

the runway, increasing the chances of collision. Of course, the more planes approaching the airfield, the more difficult it would be to coordinate them. This is where synchronization intervenes, offering a synchronous delivery of audio content to all participants.

Chapter 3

Playout buffering in minisip

This chapter provides a technical overview of the enhancements done in minisip's internal playout buffer, in order to support synchronization (see section 1.1). It also describes the syncd client/server application, which complements the synchronization process by proacively monitoring the end to end delay of the audio channels. The final section of the chapter presents a mathematical model for estimating the location of the peers based on the delay of sound propagation.

3.1 Software architecture for audio synchronization

3.1.1 Processing incoming RTP packets

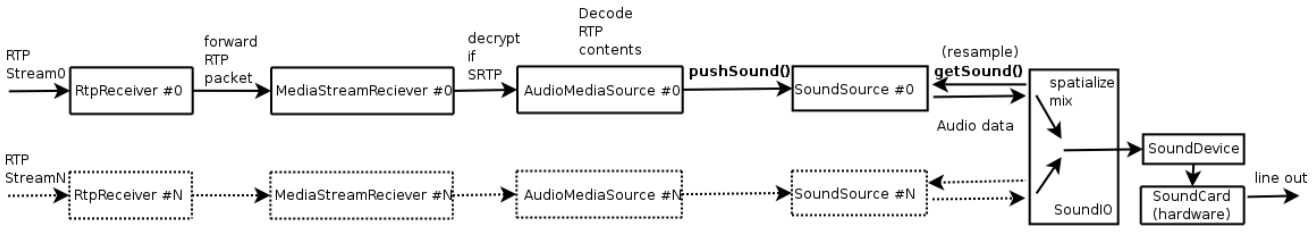


Figure 3.1: Playout procedure in minisip. Every box represents an instance of the class. The name of the class in given in the caption inside the box. The #x notation represents an instance of the class with number x. Minisip can support multiple incoming streams if operating in conference mode.

This section outlines the process of retrieving, storing and playing the payload of RTP packets in minisip. Figure 3.1 illustrates the block components of this process. Given minisip's object oriented nature, every component represents an object instance. The component's caption is the class name. Upon initialization of the program, a MediaHandler object is created. MediaHandler runs endlessly and is responsible for creating RTP sessions, after a SIP session has been established. RTP sessions are handled by instances of the RtpReceiver class. In a one-to-one conversation, one RtpReceiver is sufficient (assuming only audio is being extracted between these parties). However, when conversing with more than two participants or when there is more than one media stream per party, the multiple RtpReceiver instances are created as new threads, in order to accomodate the incoming streams. When a complete packet is read, RtpReceiver instantiates a MediaStreamReceiver, which decrypts the packets, if Secure RTP (SRTP) is being used [36]. Packets are then forwarded to an AudioMediaSource instance, which decodes the data contained in the RTP packet payload, using the PT, or payload type value in the packet's header. PT, is a 7-bit value which identifies the CODEC used in the sender to encode the audio. AudioMediaSource chooses the corresponding CODEC from the local CODEC list, and decodes the packet. Apart from decoding, AudioMediaSource also controls the bufffering process.

Figure 3.2 shows this process in detail. As soon as the data from the packet are decoded, AudioMediaSource creates a BasicSoundSource instance, which stores the data in a CircularBuffer object using the pushSound function. The CircularBuffer class implements a static array of short integers (note that we are assuming the use of 16 bit linear representations of audio samples in this buffer), and provides a basic set of access functions for reading and writing to the array. Before writing the data, pushSound checks whether the samples are stereophonic. If not, it replicates the single channel into the left and right channels before writing to the buffer. Thus all audio which is stored in the circular buffer can now be treated as if it is stereo. PushSound writes to the buffer, as long as data are available from AudioMediaSource. It maintains a write index pointer (writeIdx), offsetting it by 4 bytes (two 16 bit units) in every write operation.

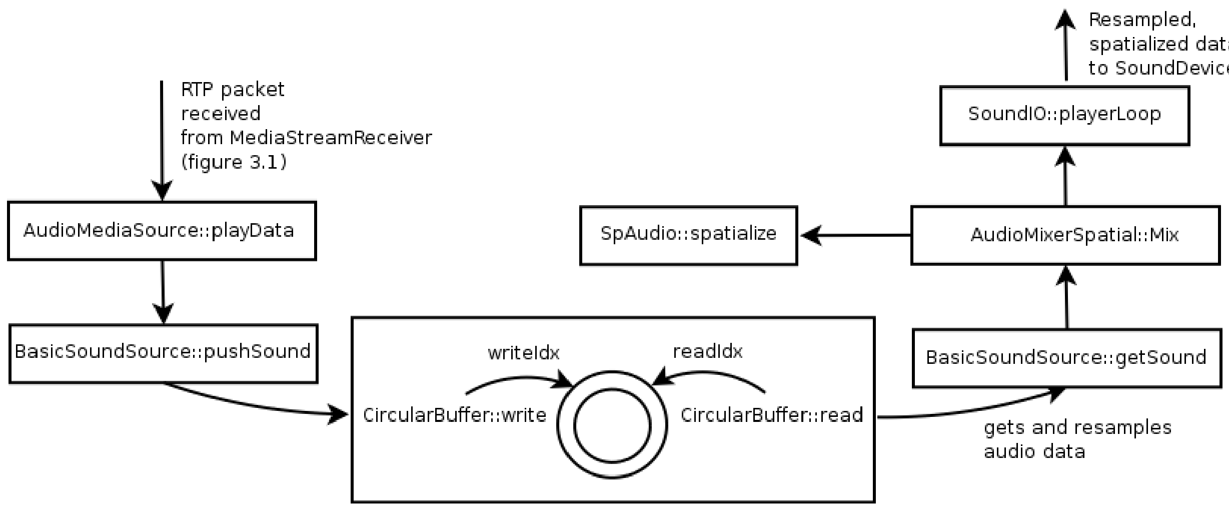

RTP packet received from MediaStreamReceiver (figure 3.1)
AudioMediaSource::playData
BasicSoundSource::pushSound
writeIdx
readIdx
CircularBuffer::write
CircularBuffer::read
gets and resamples audio data
BasicSoundSource::getSound
AudioMixerSpatial::Mix
SpAudio::spatialize
SoundIO::playerLoop
Resampled, spatialized data to SoundDevice

Figure 3.2: Minisip's internal buffer is used to temporarily store incoming audio data, before they are processed and sent to the soundcard. This picture shows the function calls that pass on the data from one object to another. Each rectangular box represents an object instance. The caption inside the box, is written in the familiar c++ style of *classname::function*.

BasicSoundSource is also responsible for retrieving audio data from the buffer using the getSound function. As with pushSound, getSound maintains a read index (readIdx), pointing to the next chunk of data, after a successful read operation. Data are then put into a temporary buffer and sent to the Resampler, for upsampling (currently conversion from 8Khz to 44.1Khz is supported). SpAudio spatializes the data according to a predefined spatialization model, which involves fixed positions in a grid around the listener [9]. SpAudio complements SoundIO, which mixes the different BasicSoundSource streams into one (in case of a conference call). Resampled data are then sent to the SoundDevice object which in turn sends them to the SoundCard for playback. In this processing, the part of concern in this thesis is the CircularBuffer storage, and in particular the relative position of the read and write index pointers.

3.1.2 Adding playout delay.

In order to keep the streams synchronized, the buffer must be flexible enough to delay the playout as long as required (see section 2.5.1). In order to familiarize the reader with the buffering mechanism, this section outlines the buffer operation without adding any additional delay.

Initially, readIdx and writeIdx point to the same place in the buffer. When an RTP packet arrives, it traverses the components described in section 3.1, and its payload is eventually written into the buffer. WriteIdx is offsetted to point to the next free location in the buffer. ReadIdx points to the newly written data and is used by getSound to retrieve the data from the buffer. After the data are read, readIdx is also icremented waiting for the next block of audio data to be written. In general, it is not to the same position as writeIdx, since the processes are running at different rates and the RTP packets are not arriving at a synchronous rate.

According to the G.711 audio codec specification which this study uses, the source sends an RTP packet every 20ms (see section 2.3). Therefore, given an uncongested network link (where none of the issues described in section 1.2.3 are present), packets arrrive orderly and on time at the receiver, every 20ms (see also experiment A in section 4.2). A 20ms timing quantization is also applicable to the functions getSound and pushSound described above: 160 samples are written and read every 20ms. The audio payload decoding procedure in AudioMediaSource, stores every sample in 2 bytes of memory[1]. Hence 20ms of incoming audio, translates to 320 bytes written to the buffer (see figure 3.3). The desired delay is introduced by offsetting writeIdx. Although this technique may seem straightforward, it actually breaks down to a number of possible scenarios, which are described separately below.

[1]The size of a short integer is typically 2 bytes, but may vary with the system architecture and data model. For all the machines we did our measurements with (see chapter 4), the GNU C++ compiler allocated 2 bytes for a short int variable. This provides sufficient storage for our 16 bit linear encoded audio samples.

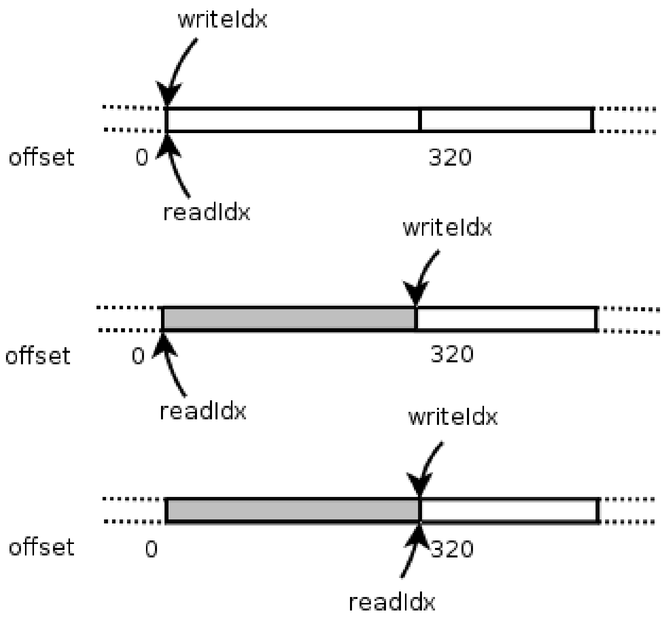


Figure 3.3: 320 bytes of audio data are written and read from the buffer every 20ms, if the network link delivering the data has no jitter.

3.1.3 Adjusting the delay before a call

The simplest scenario concerns manual adjustment of the delay before engaging in a conversation. This can be the case if the network link delivering the audio data is uncongested, or has a stable end-to-end delay.

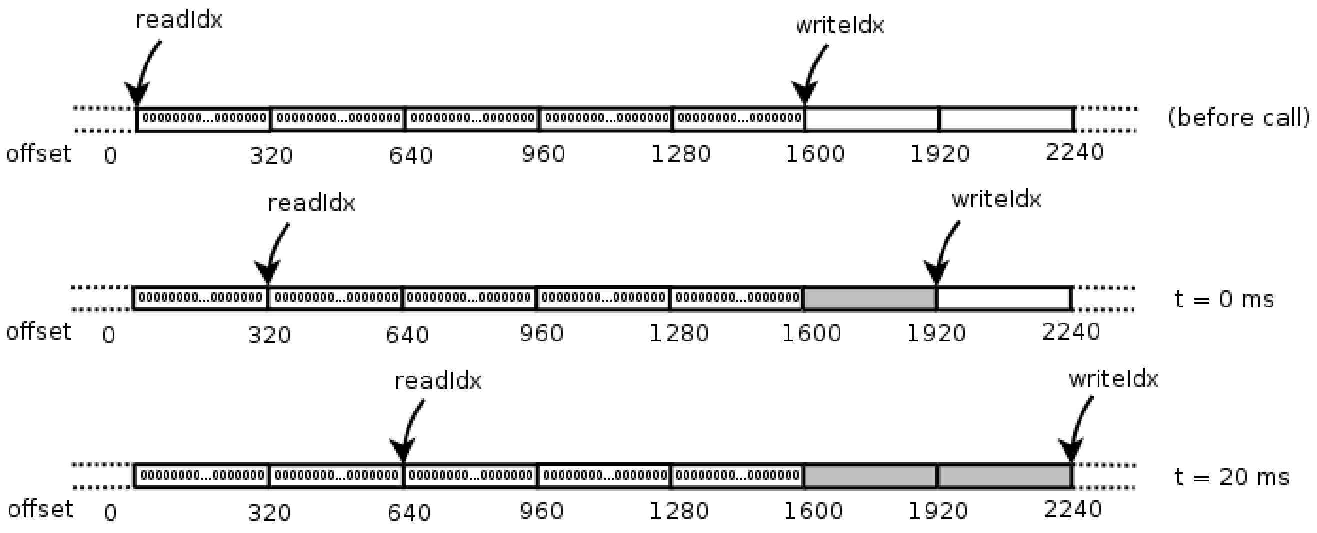

readIdx
writeIdx
(before call)
offset 0 320 640 960 1280 1600 1920 2240
readIdx
writeIdx
t = 0 ms
offset 0 320 640 960 1280 1600 1920 2240
readIdx
writeIdx
t = 20 ms
offset 0 320 640 960 1280 1600 1920 2240

Figure 3.4: WriteIdx is offseted before the call by 1600 bytes, and the [0, 1600] buffer space is filled with zeros, so when readIdx reads this data, the result will be a playout of silence.

As noted above, an uncongested network allows for timely delivery of RTP packets, since there is no of jitter or packet loss. A client receiving audio on a such an error free link probably has to wait in order to synchronize with its peer on what might be a congested link (section 1.3.2). Since readIdx acesses the buffer in quanta of 20ms, or 320 bytes, writeIdx has to be offsetted in multiples of 320. Accordingly, the delay will be increased in multiples of 20ms (figure 3.4). It is also useful to equate the desired delay(*targetdelay*) with the number of bytes the writeIdx pointer has to be offset (*offsetindex*).

$$packetpayload_{G.711} = 160samples * 2bytes = 320bytes$$

$$offsetindex_{G.711} = \frac{targetdelay * packetpayload_{G.711}}{packetizationtime_{G.711}} = \frac{320 * delay}{20} = 16 * delay$$

The type above returns the offset value in bytes, for a delay given in ms for codec G.711. For instance if a delay of 1sec is desirable, then the index pointer has to be offsetted by 16.000 bytes from its initial position. A similar technique is employed in case the network link has a fixed amount of end-to-end delay. When offsetting writeIdx, the preexisting amount of delay must be taken under account.

$$totaldelay = targetdelay * channeldelay$$

$$offsetindex_{G.711} = \frac{totaldelay * packetpayload_{G.711}}{packetizationtime_{G.711}} = 16 * totaldelay$$

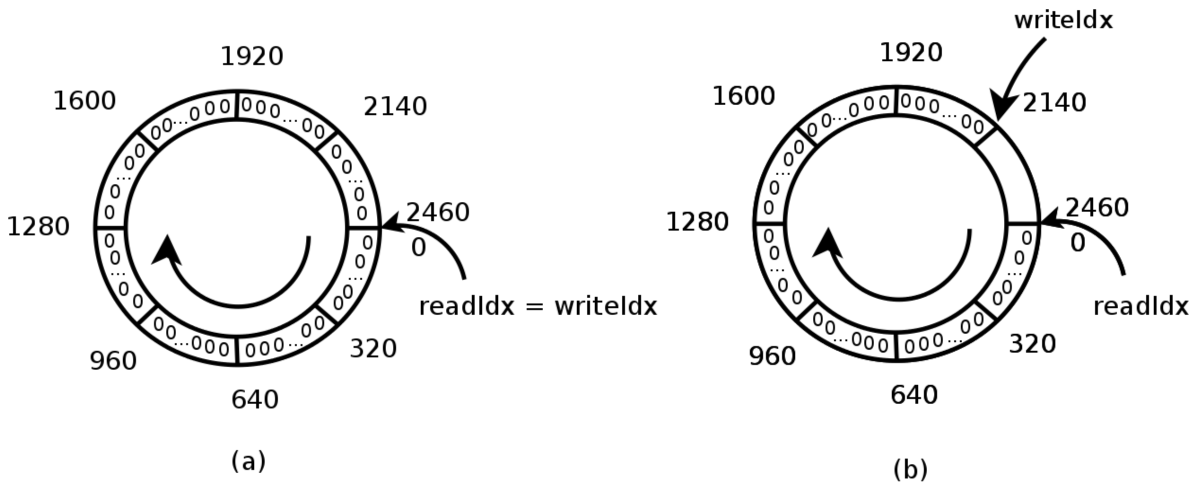


Figure 3.5: (a) The initial position. (b) To achieve maximum delay writeIdx must always be ahead of readIdx. Note that both reading and writing advance in the direction indicated by the arrows.

Another issue concerns the maximum delay possible to support given the size of the buffer. If writeIdx is offsetted to a value equal to the buffer's size (*buffsize*), it will return to the initial position, given the circular nature of the buffer. This will have no effect in delaying the playout since both readIdx and writeIdx are pointing to the starting position (see figure 3.5 a). However, offsetting writeIdx to a position of 320 bytes behind readIdx achieves the maximum playout delay (see figure 3.5b).

$$maximumdelay = \left(\frac{buffsize - 320}{320}\right) * 20 = \left(\frac{buffsize}{320} - 1\right) * 20 = \frac{buffsize}{16} - 20$$

For example, a buffer size of 32.000 bytes supports up to 1.98 seconds of playout delay.

3.1.4 Adjusting the delay during a call

In this case, the delay due to the network link changes during a call. We will begin by considering a simplified scenario which involves gradual but steady fluctuations of channel delay. A possible scenario involves three clients, S1, D1, and D2. Client S1 (the audio source) sends the same audio stream to both D1 and D2 (the two destinations for this audio). The link (S1, D1) has no errors and no jitter, whereas the link (S1, D2) experiences slow delay fluctuations. An initial delay of 20ms, gradually becomes 40ms after 100ms of conversation time have elapsed. Initially, both links are error free and have no jitter, the RTP packets are delivered on time and the audio playback from both clients are synchronized and smooth (see figure 3.6a). After 100ms, the link (S1, D2) experiences an additional 20ms of end-to-end delay, which affects packet delivery, thus client D2 receives no packets for 20ms. As a consequence, client D1 must offset his writeIdx poiter in order to stay synchronized (see figure 3.6b). This operation creates a gap in the playout. If this empty space is filled with zeros, as was the case in the previous section, it will rsult in a sound gap during the playback (a phenomenon likely to be detected by the listener). In order to create the illusion of continuity, this gap is filled with data previously received. In this example, the gap is filled with data from the previous RTP packet (conceptually named packet ”A”, as shown in the figure).

This technique closely resembles the principles on which some playout buffering mechanisms solve jitter problems in congested network links (section 1.3.3). Naturally, the greater the delay fluctuations, the more likely that the resulting disturbances in the audio are to be noticed. Our experiments indicated that this mechanism can guarantee smooth audio playback for delay changes of the magnitude of hundreds of milliseconds (section 4.3).

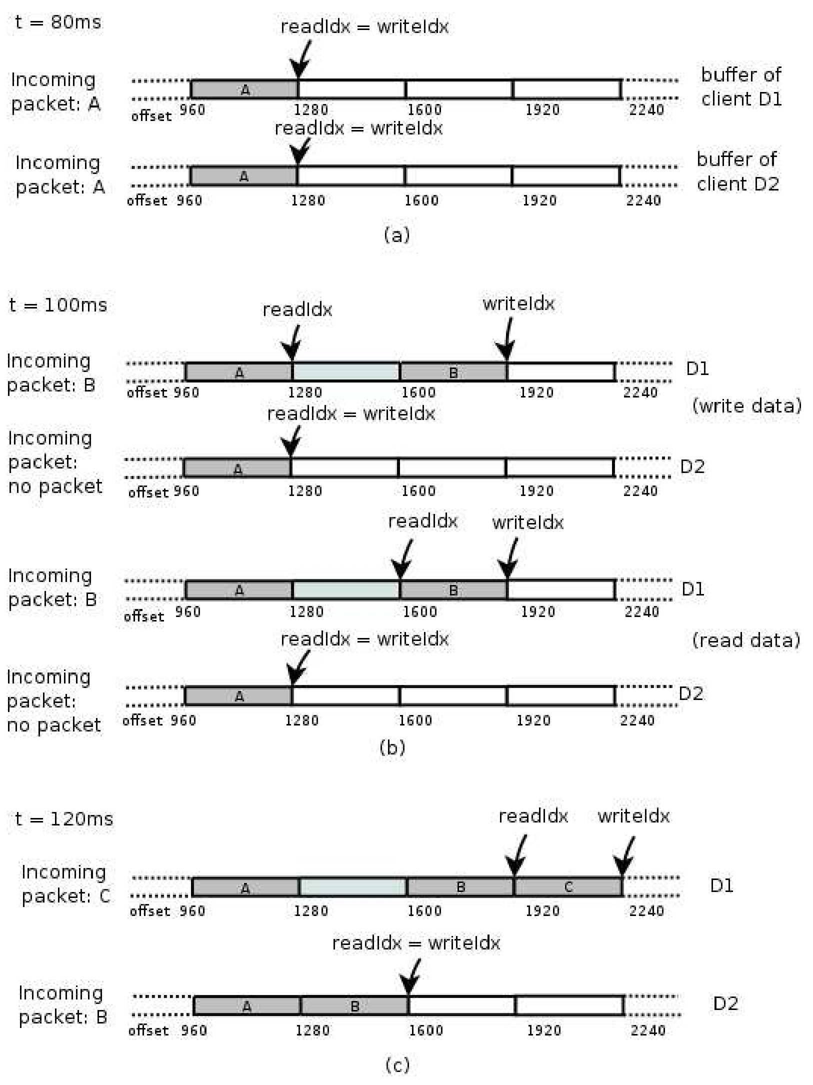
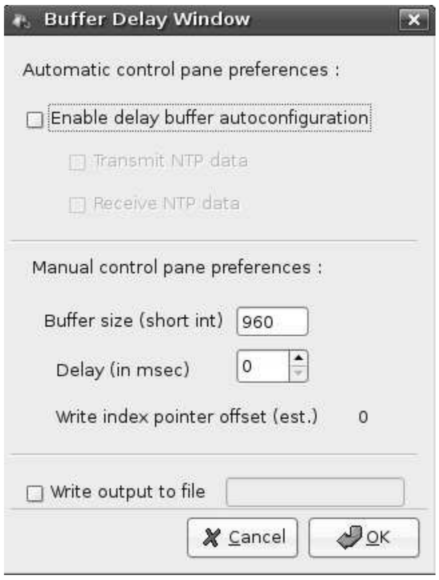


Figure 3.6: Adapting playout to changing delay.

3.1.5 Buffer delay control dialog

In order to provide easy access to the functionality implemented for the scenarios referenced in the two previous sections, a graphical control window was created. This window is integrated with the rest of the minisip GUI, providing the user with a handy tool to adjust the playout delay of the client. Figure 3.7 illustrates this window at runtime. The user can either delegate the delay adjustment responsibility to the client, or manually adjust the desired delay. The automatic configuration pane enables automatic synchronization through comparison of local NTP timestamps with NTP timestamps received from the other client(s). This is useful to handle fluctuating delay **during** a call (see section 3.1.4), and runs the syncd client/server mechanism. The client has the option to provide feedback (transmit NTP data), accept feedback (receive NTP data), or both. Syncd packets with NTP data are broadcasted to the local network and propagated to the other participants (see section 3.3). Note that this assumes that there is a local network for communication between the


Buffer Delay Window

Automatic control pane preferences :

☐ Enable delay buffer autoconfiguration

☐ Transmit NTP data

☐ Receive NTP data

Manual control pane preferences :

Buffer size (short int) 960

Delay (in msec) 0

Write index pointer offset (est.) 0

☐ Write output to file

Figure 3.7: Delay control window.

clients and that this network has relatively low delay (since it is assumed to be local). With the help of the manual configuration pane, the user can set the buffer size and the delay manually. This is useful for adjusting the delay before a call (see section 3.1.3). If the desired delay is larger than can be accomodated by the default buffer size, the user can increase the size of this buffer. The program also reports the number of bytes writeIdx will be offsetted, based on the delay the user has typed in. For debugging and/or measurements there is also option to record to a file the details of the write and read operations to and from the circular buffer.

3.2 Detecting channel delay

Syncd is a client - server protocol that carries NTP timestamps encapsulated in UDP messages, between peers. Note that the name syncd used in this thesis refers to the particular protocol described here and not to any other existing protocols of the same name. A syncd packet size is 68 bytes (42 bytes of header, and 26 bytes of payload). The payload is essentially a timestamp from the sender, in the following format: *YYYY-MM-DD HH:MM:SS.milliseconds.microsecods*. Figure 3.8 illustrates a typical session between a syncd server and a syncd client.

This figure will be described from left to right and top to bottom. The syncd_client on S1 extracts timestamps from the kernel clock and writes it to outgoing UDP packets. Syncd_server on D1 receives the timestamps and compares them to timestamps of its own clock. The difference in time between the timestamp and the local clock is a delta written to a list. When a write operation takes place in the minisip buffer, the mean value of all the timestamps of the list is computed in

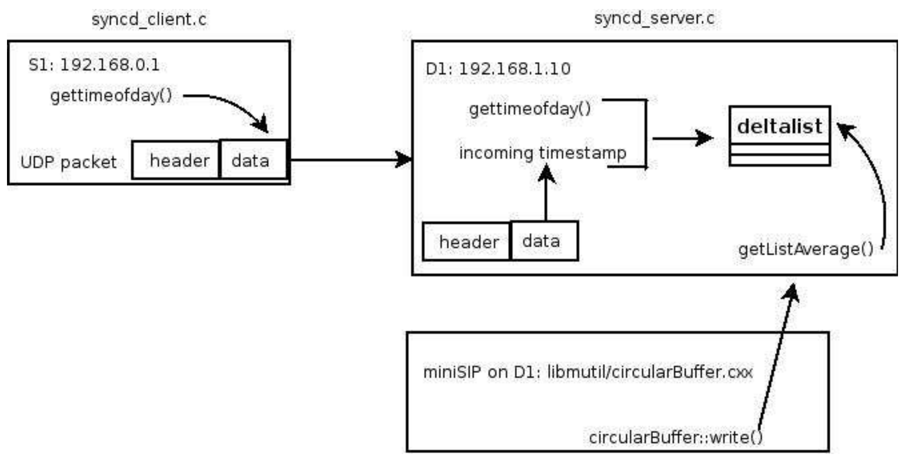


Figure 3.8: Syncd mechanism in operation.

order to adjust for the delay. The operation of syncd depends upon the assumption that the clocks of all participants are synchronized to a single source. Thus the local deltas represent the one-way delays from this source.

The number of samples can be set by the user depending on the current channel conditions (e.g. packet loss, delay, etc.), On an uncongested channel, the list is polled periodically, every 20 seconds (plus/minus one), but this periodicity is not true in other cases, as shown in the preliminary measurements of section 4.1. The client becomes aware of the channel delay by comparing the arriving timestamps, versus the global NTP clock. This information can be combined to determine the relative delay between two (or more) audio channels.

3.2.1 Comparison of RTCP and syncd

This section compares the syncd packet architecture with a previous RTCP implementation for minisip. The reader should note that RTCP is currently supported only on win32 and Pocket PC/ARM architectures. In addition, only Sender Reports (SRs) are implemented [35].

a. Packet size

Similar to syncd, SRs provide feedback about end-to-end delay of the network path, by carrying NTP timestamps from the sender. In addition SRs store the total number of RTP packets and total number of bytes of RTP payload transmitted by the sender. Although this helps in identifying packets losses, it does little to aid our synchronization mechanism. Additionally, the size of each SR packets is 94 bytes. A syncd packet is only 68 bytes, and can still be smaller, should the user chooses to report a smaller timestamp. However, the actual difference in overhead is small and should not affect the RTP stream unless the bandwidth is very low - in which case there are likely to be other problems due to high variance in link properties if there are other sources of network traffic.

b. Intertransmission time

For both architectures, the interval between two successive transmissions is a tradeoff between accuracy and bandwidth consumption. In RTCP's case, this interva is regulated by the bandwidth of the RTP stream and the number of senders participating in the converation. The idea is that the bandwidth consumed by RTP remains stable, regardless of the number of participants. This happens naturally because only one or two persons can speak at a time in conversation. Because of this, RTP is characterised as "self limiting". However, RTCP does not have these self limiting properties, since control packets are sent whether senders transmit RTP packets or not. The RTCP RFC [29] mandates that RTCP control traffic adjusts its intertransmission interval, so that it retains a fraction of the session bandwidth in relation to RTP (typically this is 5%). If SRs are sent at a coonstant rate, RTCP traffic would grow linearly with the number of (active) participants.

Syncd on the other hand, allows the user to set the interval, with an accuracy of milliseconds. While this is ideal for monitored environments, such as the testbed in our lab (see section 4), it is not suitable for more widespread use as it is not scalable.

In conclusion it should be noted that syncd was designed as a temporary solution to provide clients with information needed to synchronize their playouts. The proof of concept synchronization mechanism as implemented by syncd, should adopt RTCP when it becomes available for all the minisip platforms (i.e. architectures). RTCP is an established standard which allows for wider deployment of playout buffering due to its scalability. In addition it guarantees interoperability with other clients, and provides more information on the quality of the RTP stream.

3.3 Location awareness

This section presents a method for enabling clients to discover their relative positions automatically. Our test case scenario involves three clients, namely C1, C2, and C3 who want to participate in a conference call. Before the call starts, the clients undergo a training session, where each one determines its distance from the other two. Output of C2 is captured from the microphone of C2, and so is the output of C3. The result is a synthesized stream coming back to C1 (figure 3.9). C1 becomes aware of the difference in delay (for example by computing a fast fourier transform, or FFT, it is possible to decompose the waveform and to reveal the initial signals).

Our approach is simple. A simple 22 kHz tone at of 10ms duration is periodically transmitted from C1. C2 receives the original 22 kHz version and C3 uses a half-amplitude signal at 11kHz (This makes it easy for C1 to distinguish between signals, when receiving the synthesized stream

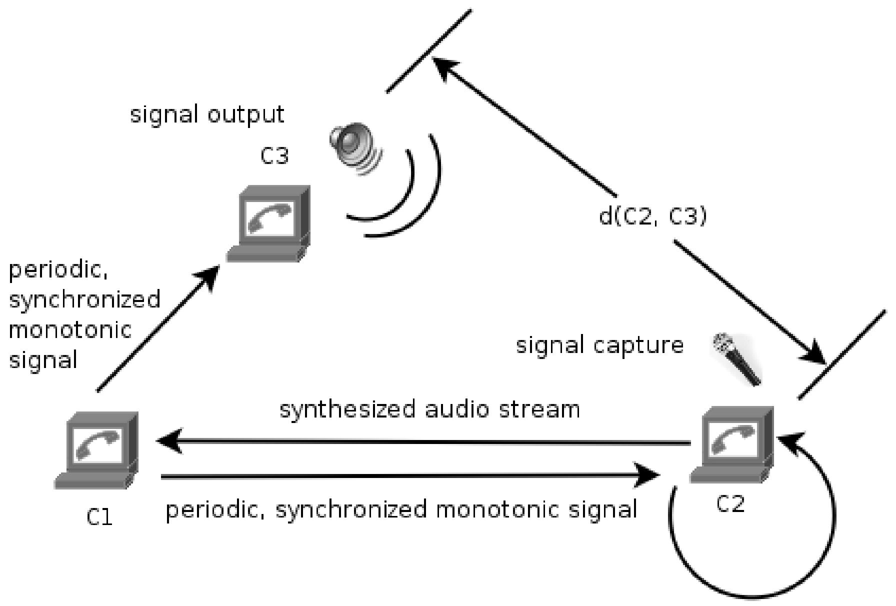

signal output
C3
periodic, synchronized monotonic signal
d(C2, C3)
signal capture
synthesized audio stream
C1
periodic, synchronized monotonic signal
C2

Figure 3.9: d(C2, C3) is the physical distance between C2 and C3. t(d) = t(C3) - t(C2) is the propagation time of the audio wave from its source (C3) to the capture device (C2).

back). Our measurements compute the differences in arrival times in milliseconds. 7 samples of (d(C2, C3), t(d)), were plotted as points in a Cartesian system (at various distances). We compute least squares of a linear regression model to fit a line to this data. Using this estimator, C1 can determine the relative position of C2 to C3, given the delay delta between the two clients (see figure 3.10).

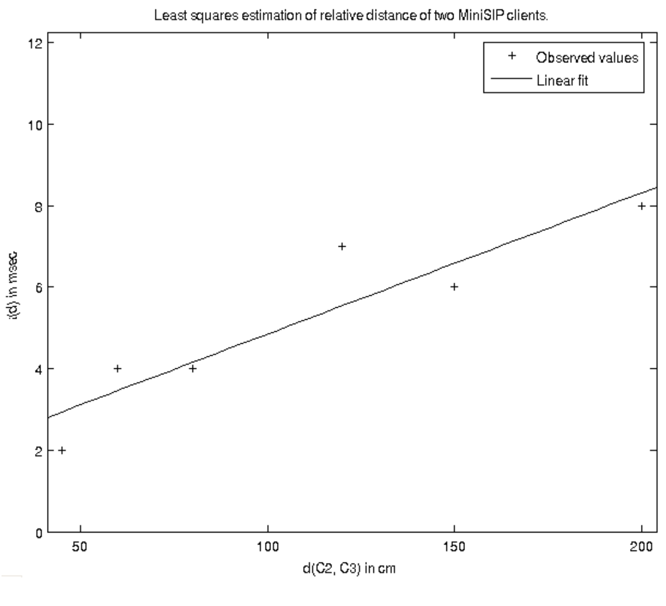


Figure 3.10: Determining the distance between the clients using a linear regression model.

Using the equation of the above linear plot, C1 can infer the distance between C2 and C3, just by comparing the relative delay of the two waveforms in the incoming synthesized stream. An approximation of this equation, using 4 decimal digits is:

$$t(d) = 0.0348 * d(C2, C3) + 1.3688 \Leftrightarrow d(C2, C3) = \frac{t(d) - 1.3688}{0.0348}$$

Assuming that the temperature and humidity remain stable in the place the conversation takes place (so that sound propagation time from C3 to C2 is not affected), then this equation can be used by the other clients as well, in order to determine their relative distance. When all 3 distances are known, a triangle is formed having clients as edges and their distances as sides.

The goal is to translate the position of the clients to cartesian coordinates, in order to construct a spatial model. This process, requires calculating at least one angle of the triangle (see figure 3.11).

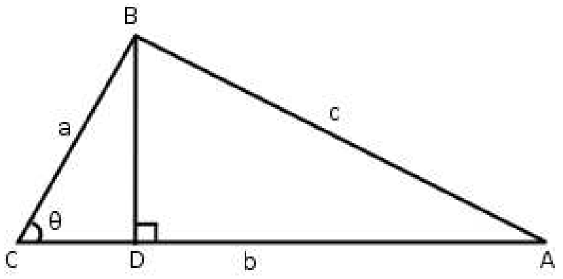


Figure 3.11: In this triangle, clients C1, C2 and C3 are translated to verticles C, A, and B respectively.

In order to find angle θ , we make use of the law of cosines:

$$c^2 = a^2 + b^2 - 2ab * cos(\theta) \Leftrightarrow cos(\theta) = \frac{(a^2 + b^2) - c^2}{2ab}$$

$$cos(\theta) = \frac{CD}{a} \Leftrightarrow CD = acos(\theta)$$

$$sin(\theta) = \frac{BD}{a} \Leftrightarrow BD = asin(\theta)$$

It is now possible to project the triangle onto a cartesian coordinate system (see figure 3.12). We arbitrarily chose point A (client C2) to be the point of reference (0.0) and plot the rest of the points with respect to it. This example demonstrates a simple mechanism for spatial audio auto-configuration for minisip. However, this method is not (yet) intended to enhance the experience of the system. The spatial audio model used in minisip is very basic and supports only 2 degrees of freedom (forwards, backwards and left, right)[2].It should be noted that this method does enable the

[2]More complex and expensive spatial audio systems are not confined in two dimensions and take under account the three dimensional space of the real world (x, y, z) for describing the exact position of the sound source. In addition, they make use of Euler angles (another set of $\widehat{X}, \widehat{Y}, \widehat{Z}$ triplets), to define the orientation of the sound source, supporting 6 degrees of freedom in total.

determination of the relative position of speaker, which might be used to locate and differentiate speakers around a table. It could also be used to provide input to acoustic echo cancellation, since you can explicitly know where the other source sources are and hence use this information in cancelling these sources out.

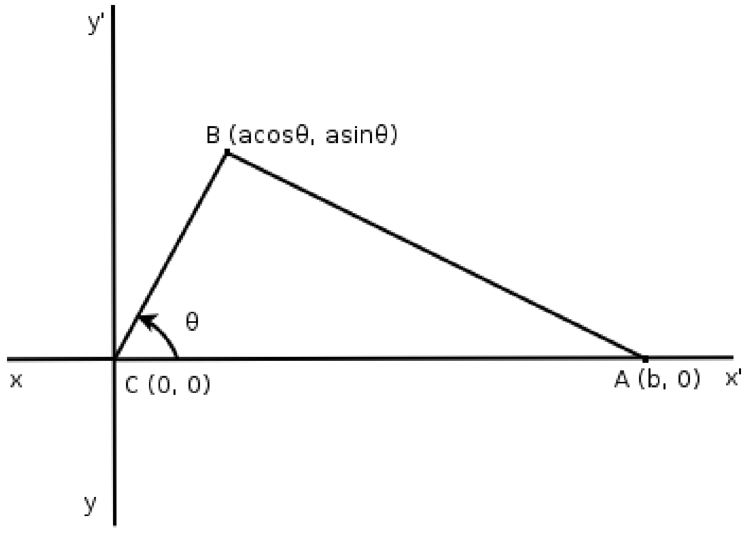


Figure 3.12: Spatial model for a 3-client conversation.

Note that if there is a tone with a known position participating in the experiment, the ambiguity of peers having relative separations is resolved. In addition, given the current temperature in the place of the experiment, it is also possible to calculate the humidity, thus obtaininng a very good measure of the comfort of people in the experiment area. This information can successively be used to control the heating, ventilating and air conditioning system to keep them comfortable

Chapter 4

Measurements

4.1 Introduction

4.1.1 Testbed

This chapter presents a set of measurements and subsequent statistical analysis conducted in order to determine the efficiency of the synchronization mechanism. The testbed setup (figure 4.1), is comprised of two subnetworks, which exchange traffic through a router.

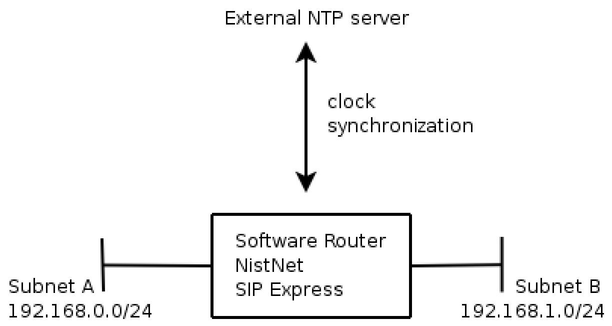


Figure 4.1: The network testbed, as used in this study.

This router is actually a computer with two ethernet interface cards (NICs) attached to it, thus all cross-subnet network traffic traverses through this machine. The reason for opting for this topology, was to emulate network links with specific performance characteristics, such as packet loss, jitter and end-to-end delay (see section 1.3.3). The network emulator we used, is called Nistnet [37]. Nistnet is a pluggable Linux kernel module, running on the router. It operates by processing all incoming network traffic. It can also differentiate between specific traffic streams, by letting the user define source and destination addresses of packets to be processed when defining rules. In addition, the router acts as an NTP server as well. Clients from both subnets synchronise to the local clock of this router, which in turn synchronizes with an external NTP server of lower stratum. Finally, the router also functions as a SIP registar, storing the IP addresses of peers, and establishing SIP sessions upon request. The SIP registar was implemented using SIP Express proxy, together with a mysql database [32].

Clients in both subnets were running minisIp, using the enhanced playout buffering mechanism described in chapter 3. In addition, they have synchronized their clocks with the local NTP server's clock (which is the same machine as the router). When in conversation, clients, were running syncd as well, so they were able to detect any network link delays between them and their peers.

The operating system for the clients was GNU/Linux, specifically the Ubuntu distribution, version 6.10. The reason this distribution was chosen is because it allows for easy installation of the minisip client. The server was running Redhat linux 9, because its kernel version was appropriate for nistnet (because of changes in the netdevice.h kernel header, nistnet does not currently work on kernel versions newer than 2.6.13.x). From a hardware persepective, the clients were two Dell Optiplex GX620 desktops with Intel Pentium-D processors, a laptop with an Intel Core Duo, and a laptop with an IBM PowerPc G4. The server, was an IBM Pentium-3 laptop, eqquiped with two PCMCIA NICs.

4.1.2 General measurement information

The measurement precision was set to 1ms for all the experiments in this chapter. In addition, the voice CODEC used was G.711 (see section 2.3).

4.2 Preliminary measurements

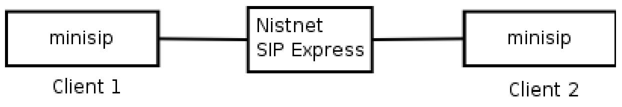


Figure 4.2: Testbed for preliminary experiments. Client 1 calls client 2 and we monitor the write index pointer in client 2's buffer.

This set of experiments was designed to find patterns in the behavior of the MiniSIP write index pointer, under different network conditions. This series of experiments was carried out prior to the implementation of the delay mechanism described on chapter 3. In particular, the amount the write index pointer has to be offsetted each time, is based on information extracted from these experiments (figure 4.2). The testbed involves two clients in a conversation, where the RTP traffic traverses through the router and is regulated by nistnet. We examine four distinct scenarios.

4.2.1 Writing to the buffer in timely manner, on a non-congested network

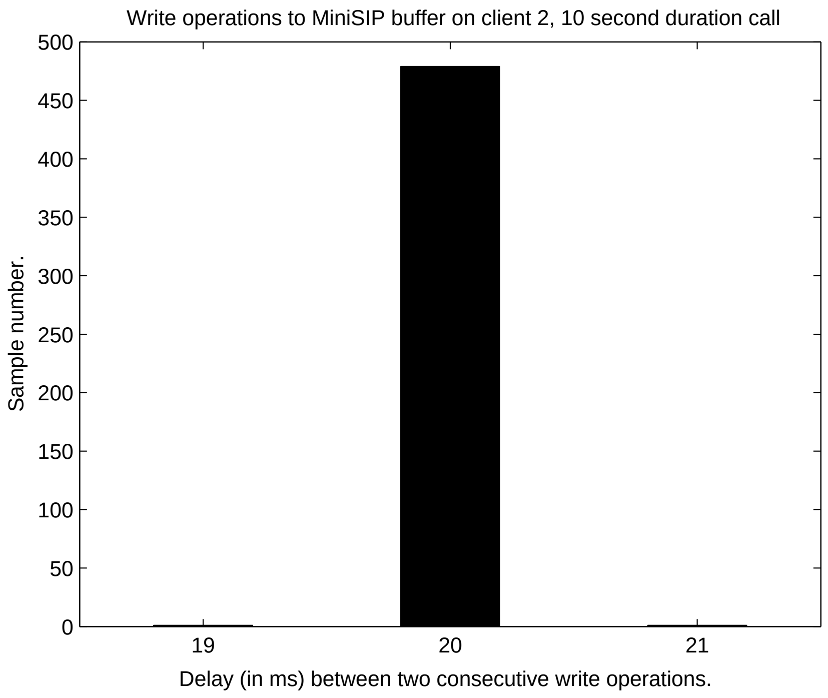


Figure 4.3: Timing between two consecutive buffer writes for a healthy network link.

In this scenario, there are no active rules running on nistnet. The conversation took place between client 1 and client 2 for a duration of approximately 10 seconds. As expected from an uncongested network, the packet interarrival time on client 2, equals the packetization time on client 1. Therefore, the payload from each packet is decoded and written to the buffer every 20ms. This observation allowed an initial evaluation of playout buffering in minisip (see section 3.1.2). Although there is a small variance of the magitude of 1 msec, it accounts for only 0.4% of the samples, and is too small to affect the result - in fact it is too small to even be seen in the figure (see figure 4.3).

4.2.2 Constant end-to-end delay

Time from start of call (in seconds)	End-to-end delay (in ms)
0-1 sec	No delay
1-5 sec	20
5-10 sec	100

Table 4.1: Gradually adding delay to the network link - intervals of constant added delay.

This experiment monitors the timing of the write index pointer on a network link with constant end-to-end delay. However, the end-to-end delay is not static throughout the 10-second call, instead, it is gradually increased, in fixed amounts. The exact timing is shown in table 4.1. The reason behind this experiment was to provide feedback for the scenarios described in section 3.1. Minisip's audio buffering infrastructure worked as expected, and stored the RTP packet payload as soon as it arrived on client 2. The nistnet emulator enqueued the incoming packets from client 1 in its internal buffer for a duration equal to the specified delay interval. After this time interval for a packet had elapsed, nistnet extracted it from the buffer and sent it to client 2. Naturally, the longer the end-to-end delay, the longer the packet interarrival time on the receiver, thus the longer the time between two consecutive write operations to client 2's buffer. This is illustrated in figure 4.4. The histogram reveals three distinct cases of buffer writes, corresponding to the timing scenario mentioned previously.

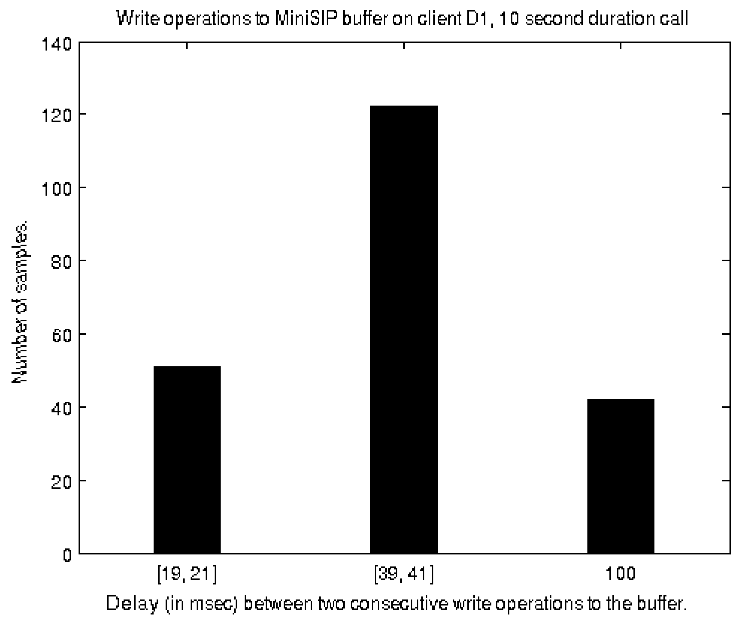
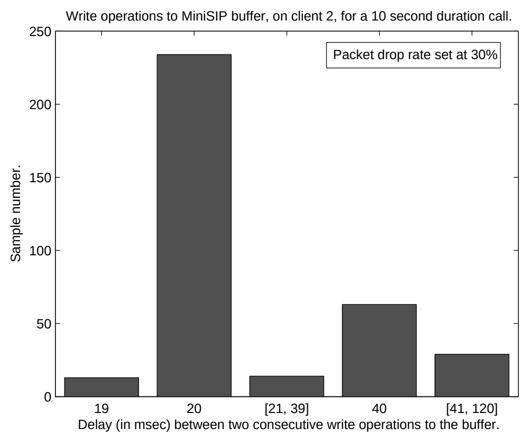


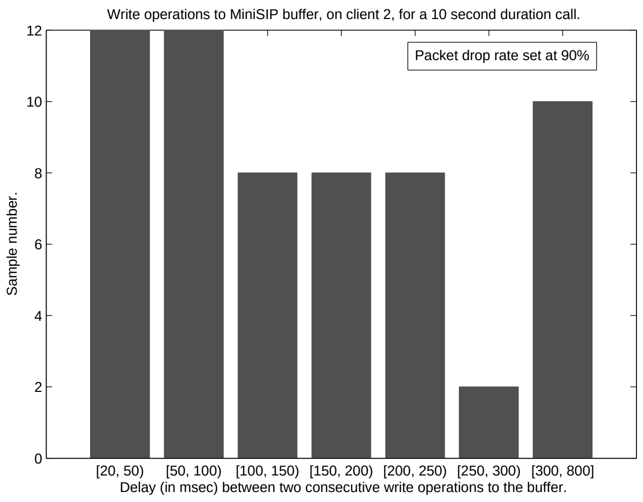
Figure 4.4: Timing between two consecutive buffer writes for a network link with variable end-to-end delay (see table 4.1). Note that [a, b] dictates the range of delay between a and b ms.

4.2.3 Packet drops

This experiment actually consists of two scenarios, one with a modest 30% packet drop rate, and another one with an extreme 90% drop rate (figure 4.5). The calls had a duration of 10 seconds for each scenario. The RTP packets that make it through to client 2 are read immediately. If a group of packets is dropped, there are no audio samples to write to the buffer so the write index pointer remains in the same place, until new data comes in. In this experiment, the idle time of the pointers can range from 20ms (in the case of timely delivery of packets) to 800ms (which signifies that a large number of packets have been dropped since the last write). It should be noted that nistnet provides no control of which packets to drop, instead it guarantees that it will arbitrarily drop as many packets as necessary to meet the percentage set by the user.



(a)



(b)

Figure 4.5: Timing between two consecutive buffer writes for a network link with 30% packet drop rate (a) and 90% packet drop rate (b). Packet drops reduce the number of writes as comparet to the ideal case of the experiment described in section 4.1.2, and arbitrary timing of to the buffer. In the case of 30%, access times manage to stay within an acceptable upper limit of 100ms. Note theat [a, b] dictates the range of delay between a and b ms.

As is the case with the minisip's implementation of the G.711 codec, no techniques are used (see packet loss in section 1.3.3), therefore, the audio playout inherently will have sound gaps, corresponding to lost packets. In this case, the synchronization mechanism cannot guarantee a gapless playback, even though the playout may be synchronous. If packet loss concealment is used, then it is possible to support stream synchronization, since some amount of lost audio data will be resynthesized on the receiver.

4.2.4 Jitter

Nistnet supports jitter, by normally distributing individual packet delays around a mean value specified by the user $(\bar{x})$. In addition, the user specifies the standard deviation of the random variable $\sigma(\bar{x})$ [1]. Figure 4.6 illustrates the PDF plot of a channel with packet jitter. Although the jitter is random, nistnet uses a normal distribution model to generate delays around the default value of 20ms. In addition, the standard deviation is used to indicate delay dispersion. For

[1]The propability density function (PDF) for the delay is selected at compile time. The user can select between normal, pareto and "experimental" distributions. The experimental distribution (default selection) is an empirically derived "heavy tail" distribution [38]. This jitter experiment uses the normal distribution.

example, a small standard deviation of the magnitude of few milliseconds (such as the one used in the example above) is an indication of low channel jitter. A larger value, indicates greater dispersion, and suggests a more unstable network link.

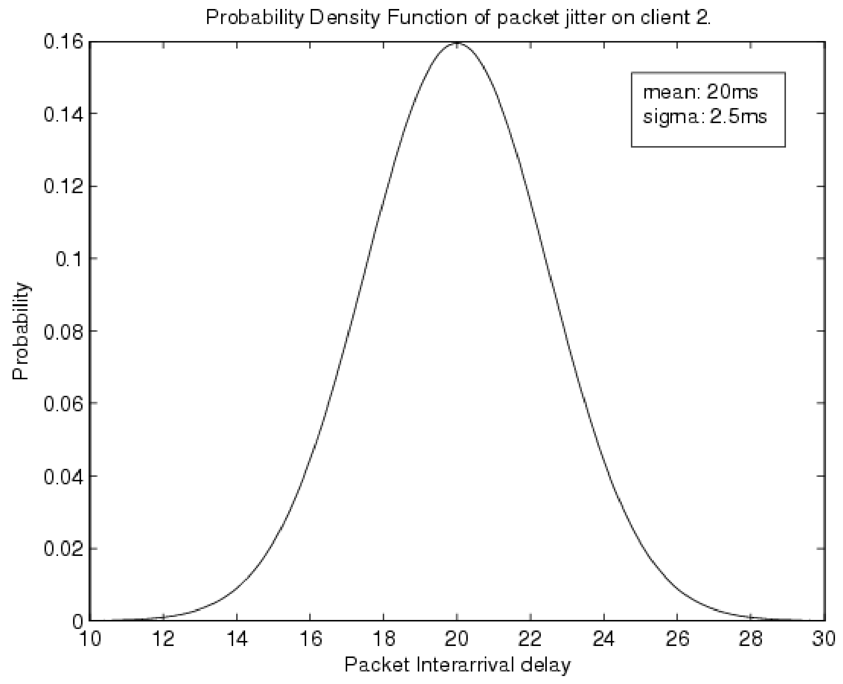


Figure 4.6: PDF of normal distribution of jitter, around a mean of 20ms with a standard deviation (sigma) of 2.5ms. Since this is a normal, bell-shaped gaussian curve, 95% of the values fall within the [17.5, 22.5] set, and 99.7% within the [15, 25] set.

The buffer in client 2 tries to accomodate the packets as they arrive. Therefore, packets arriving in bursts, will be written to the buffer immediately. In this case, the write index pointer, may overwrite previously written data that have not yet been read by the read index pointer. In addition, long delays between packet bursts dictate that the write index pointer must be offsetted in order for the playout to remain synchronized. Since the interarrival time of the packets is random, the buffering playout mechanism must choose between two possible solutions.

A possible soluton is to model the packet interarrival time and try to fit it into a distribution, such as the one used in the example above. Then, based on probability scenarios, the client must make decisions on how to buffer incoming packets. This can be difficult, partly because jitter does not always fit into a distribution, and partly because the value dispersion might be so large, that does not permit safe timing predictions for the future.

4.3 Performance evaluation.

4.3.1 Synchronizing client on a non-congested network, and gradually adding delay

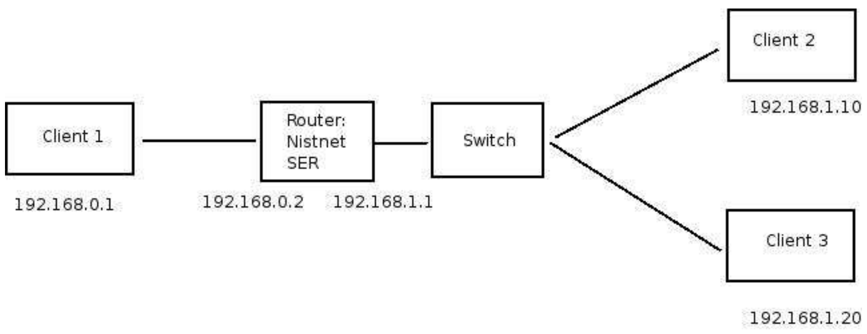


Figure 4.7: Testbed for experiment 4.3.1.

Problem statement: Client 1 sends audio data to Client 2 and Client 3 (see figure 4.7). If it is desirable to have a stable relative delay between Client 2 and Client 3, if the network conditions change (for example the Client 1 to Client 2 link receives additional delay), is it possible to maintain the playout in Client 2, Client 3 synchronized? The testbed used in this scenario is illustrated in figure 4.7. If the delay between Client 1 and Client 2 changes, we track the upper bound of this change, so that Client 2 and Client 3 maintain the same relative synchronization. In order to have a zero relative delay, the initial delay must be at least: *max(delay(Client 1, Client2), delay(Client 1,Client 3))*. We assume the local processing of audio packets in Client 2 and Client 3 is constant (as both clients run on identical hardware using identical software). We also assume that the difference in delay is primarily due to differences in transmission delay. In every write cycle on client 2 and 3:

- Syncd on client 3 gets a delay average of the (client 1, client 3) link. syncd running on client 2 gets an average of (client 1, client 2) link.
- Clients exchange their source link delays, and determine the max of the two.
- Both clients set their delay to the value of the largest of the two averages.

Timing for this scenario is shown in tabe 4.2. In the first 10 seconds, packets get delivered to Client 2 and 3 in a timely manner, almost identical to the packetization time on client 1 (20ms) (see section 4.2.1). Any delay adjustment takes place after the tenth second. We incrementally change the channel delay, with a step of 0.2 seconds, every 10 seconds, for the duration of the call. From the 10th to the 20th second, packet delay causes client 3 to adjust the timing of his write buffer, and now write every 220ms, since the data suffer a 200ms channel delay, prior to delivery.

The pattern keeps repeating for the following time spans, presenting a similar behavior to that of experiment in section 4.2.2. The question we want to answer is whether the mechanism detects the added delay in the (Client 1 and Client 3) channel every 10 seconds, and adjusts the playout by offsetting the write index pointer in both buffers, in such a way that the final result is identical to the result of the first 10 seconds, when there was a delay in the channel.

Time elapsed (in seconds)	(client1, client2) channel end-to-end delay (in ms)
0 - 10 sec	No delay
10 - 20 sec.	0.2 sec.
20 - 30 sec.	0.4 sec.
30- 40 sec.	0.6 sec.
40 - 50 sec.	0.8 sec.
50 sec - 1min..	1 sec.

Table 4.2: Timing on experiment 4.3.1.

For empirical verification, we made use of a speech audio file, encoded in stereo, at 44.100 Ksamples/sec. Client 1 sends the same audio data from the same file to Client 2 and 3. In order to verify that the clients remain synchronised we combine the audio from the right channel of client 2, with the left channel of client 3.

4.3.2 Empirical evaluation

Our first observation concerns some small inaccuracies in the syncd mechanism, which uses UDP packets to carry NTP timestamps, in order to enable the receiver to measure the channel delay. Based on this information the receiver offsets the buffer. Therefore, we will examine what happens in the case of a transition from a state of no channel delay to a state of 0.2 sec channel delay on the 10th second.

Issue A: When the 10th second occurs, nistnet is scheduled to increase the delay of the Client 1 to Client 3 channel by 0.2 sec. Such a delay is universal; i.e., it applies not only to the RTP packets carrying the audio data, but to the NTP data as well. This means that the first UDP packets with NTP timestamps sent after the 10 sec. mark, will not arrive to client 3, before an additional 200 ms have elapsed. Meanwhile, client 2 will continue playback until notified by client 3 that its buffer must be offseted as well. Then both clients will offset their buffers. The empirical interpretation of this in our lab was that the left speaker continued playing for 200 ms, while the right audio speaker did not produce any sound, because there were no audio data coming in.

Issue B: There is also the issue of the packetization time of the UDP packets (this can be set by tweaking the TR_PERIOD macro in syncd_client, see appendix B.2). A good convention is a time of 1ms. This would provide 20 samples of the channel delay for each RTP packet, enough to make a highly accurate (5% error) decision for the next step.

4.3.3 Conclusion

The experiment described above, helped us to reach some meaningful conclusions concerning our synchronization mechanism:

- If the relative delay changes during a conversation,
  - Rapid and large changes of one channel delay with respect to the other, will result in large relative delays, which can lead to sound gaps during playout. These sound gaps can be filled with previous sound data, in order to maintain smooth playback.
  - For relative delay changes under the threshold of 100ms, the replay of previous sound data are unlikely to be noticed by the listener.
  - For relative delays greater than 100ms, a secondary FIFO buffer must be used, in order to maintain smooth synchronization (see section 4.2.4).
- If the relative delay is kept steady during the conversation,
  - We can deal with any stable relative delay throughout a conversation, as long as the buffer size is sufficient.

The reader should note that this experiment was empirical (see figure 4.8), in other words it was based on a subjective perception of sound. The thresholds above may vary based on differences in the hearing of different people. Our aim was to provide a generic solution that was not completely accurate, but could synchronise playback without the need of special hardware or software.

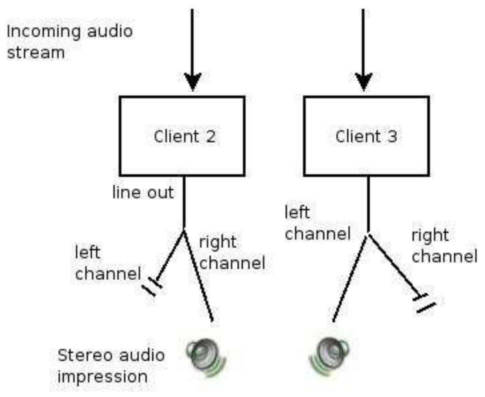


Figure 4.8: Evaluating synchronized playout buffering based on empirical perception of stereophonic sound.

Chapter 5

A Pocket PC port for minisip

5.1 Basic concepts

5.1.1 Evolution of the Pocket PC platform

Microsoft's Pocket PC (also referred to as P/PC or PPC), is an operating environment for a type of handheld device known as personal digital assistants (PDAs), running a specific version of Microsoft's embedded operating system known as Windows CE. Pocket PC PDAs all have a touch-sensitive screen as a common input medium, while some feature an additional set of buttons to further accomodate user input. Although PDAs running Windows CE dated as early as 1996 (with the introduction of Windows CE version 1.0), it was not until 2000 that the first Pocket PCs came to market. The Pocket PC platform imposed strict guidelines on PDA manufacturers and was specificaly targeted to compete with the established Palm OS based PDAs. Table 1 lists the several iterations of the Pocket PC platform from its introduction in 2000, until today.

Platform	*Version of Windows CE*	*Introduction date*
Pocket PC 2000	3.0.9348 (Build 9351)	April 19th, 2000
Pocket PC 2002	3.0.11171 (Build 11178)	June 9th, 2001
Windows Mobile 2003	4.20.1081 (Build 13100)	June 23d, 2003
Windows Mobile 2003 (Second edition)	4.21.1088 (Build 14049)	March 24th, 2004
Windows Mobile 5.0	5.1.1700 (Build 14334 to 14397)	May 10th, 2005

Table 5.1: Versions of the Pocket PC platform.

The introduction of Windows Mobile 2003 was a landmark for the Pocket PC platform. With Windows Mobile, the independent Smartphone and Pocket PC platforms supported by Microsoft, converged into one unified platform, under the .NET compact framework (.NET CF). The .NET CF allowed the developers to create applications that could run on any mobile device, thus avoiding the earlier requirement to write different versions for different platforms. In addition Windows Mobile 2003 added support for IEEE 802.1x network access control. Microsoft also provided Visual Studio (VS) .NET, an integrated development environment for authoring appliations for Windows Mobile 2003 enabled devices. VS 2005, was the successor to VS .NET and added support for Windows Mobile 5.0. The port described in this chapter was developed in VS 2005, and supports both the Pocket PC 2003 (Windows Mobile 2003) and Pocket PC 2005 (Windows Mobile 5.0) platforms.

5.1.2 The operating environment of a Pocket PC device

When designing applications for the Pocket PC, the programmer must take under account certain unique characteristics of the platform, that do not apply to traditionall interface design for desktop computer systems. The most prominent distinction is the resolution of the display, as measured in pixels[1] . Today, the most commonly used resolution for desktops is 1024 by 768 pixels, this is typically displayed upon large displays (15 inch diagonal viewable image size at the minimum) that can make use of this resolution. In comparison, Pocket PCs have smaller displays, due to their handheld nature (some common sizes are 3.8, 3.5, or 3.0 inches of diagonal viewable image size). According to Microsoft, the minimum resolution a Pocket PC must support, is 240 by 320 pixels. There are devices that support higher resolutions (most common of them being 240 by 640 and 480 by 640 pixels), however in order to facilitate use on a wide range of devices, an application should be designed for 240 x 320 screens, but should adapt itself to higher resolution displays when they are available.

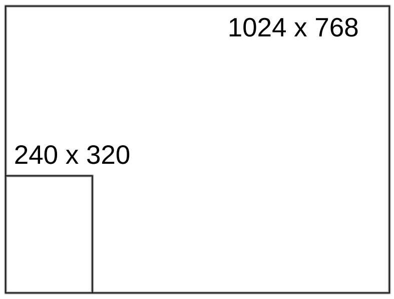


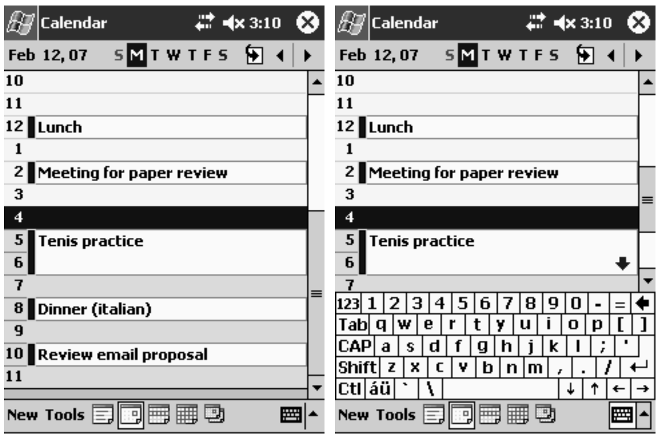
Figure 5.1: Comparison of viewable area for a common desktop PC monitor and a minimal Pocket PC display.

Figure 5.1 illustrates the significant difference in resolutions. The problem of designing an application for Pocket PC, becomes even more complex if the application has a preexisting graphical user interface (GUI) designed for a desktop system, as is the case with minisip. In order to replicate

[1]A pixel is an abbreviated term for picture element, and is conceptually defined as the smallest element of an image. Naturally, the more pixels, the more accurate the representation of an image on a display. Display resolution is often described in terms of the number of pixels per row and column (for example, 240 by 320 or 240x320).

the functionality of the desktop GUI, careful attention must be given to the UI, in order to scale down the interface to fit the small screen of the handheld device, while maintaining the same levels of functionality and ease of use - as well as maintaining the user's familiarity with the interface - since a GUI that is somewhat different requires longer to adapt to than one which is much different.

Another differentiating feature of Pocket PCs from desktop computers, is the method of user input. The input medium on Pocket PCs is a stylus, and occasionally one or more buttons mapped to specific functions (known as softkeys). Apart from a few exceptions, most devices are not equipped with full-size keyboards. Instead, the main medium for text input is a virtual keyboard known as soft input panel. The user taps with the stylus on the keys of this panel in order to enter text. However, the soft input panel is cumbersome and tedious to use over the long term. In addition, when in use, it covers a large fraction of the screen, thus hiding information that can be of use (see figure 5.2). Thus minimizing required text input from the user is another requirement for a Pocket PC application. Section 5.2 outlines the usabillity and functionality shortcomings that became evident when scaling the minisip's Windows x86 GUI to the Pocket PC, as well as the solutions employed to overcome these problems.



(a) (b)

Figure 5.2: (a) A clutter-free screen with today's appointments. (b) When the user wants to enter a new appointment using the soft input panel, the limited viewing area prevents the whole information to be presented in one screen - thus the user has to resort to using the scrollbar on the right. In addition the virtual keys of the panel are too small and the user is prone to make mistakes while typing. This example shows the calendar application, built in since Windows CE 4.

An additional obstacle for porting applications originally written for desktop operating systems (GNU/Linux and Windows x86 in the case of minisip), concerns the limitation of available libraries in the .NET CF. Prominent examples include the lack of support for the Standard Template Library (STL), the open-source secure sockets layer library (OpenSSL), etc. Before planning to port an application, all the library dependencies have to be reviewed, and if necessary, any missing libraries need to be ported to the Pocket PC platform. This is exactly the methodology followed when porting minisip. Section 4.3 describes the technical aspects of the porting process in more detail.

5.2 Design and implementation of a GUI

5.2.1 Basic principles

The GUI for Pocket PC devices was designed using a set of guidelines deemed necessary in order to maintain the usability and functionality of the version previously developed for dekstop systems. The fundamental requirement was to maintain the number of options available in the original GUI. In addition, this should be done in a way that minisizes text input and reduces information clutttering per application window.

5.2.2 The dial window

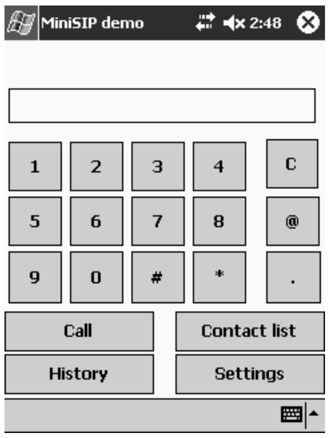


Figure 5.3: The main window of minisip's GUI for Pocket PC with a basic keyboard for entering SIP URIs.

Figure 5.3 illustrates the main window of the application, as shown on the screen after minisip is launched. From this windows, the user has the option to place calls, either by entering a SIP URI[2] or using a contact list of previously registered peers.

A SIP URI's general form is: *sip:user:password@SIPproxy:port*. Normally when making a call, a user would specify the username and the proxy of the person they wish to call. If the username consists solely of numeric characters, and the user knows the IP address of the proxy, then the SIP URI consists only of numeric characters and the special character "@" (for example:

[2]URI or Universal Resource Identifier is a string of characters used to identify a resource. The actual format of the string specified by the protocol that uses it. URI schemes are mainly used in markup languages such as Extended markup languate (XML), hypertext markup language (HTML), and Microsoft's Extensible Stylesheet Language Transformations (XSLT), but they are also used in SIP, to identify a peer. SIP proxies translate SIP URIs into IP addresses.

1234@192.168.0.1). In this case, the user can place the call using only the keyboard provided, instead of having to resort to the soft input panel, which requires a combination of keys to produce the "@" character. However, as this is a personal device, the user is most likely to want to contact someone they already have some relationship to, hence this contact is probably already in the user's contact list. In this case, the user simply has to tap on the "Contacts" button on the lower end of the screen (section 5.2.3).

The "Settings" button brings up the user settings screem, allowing tweaking of various program settings (section 5.2.4), or user profiles (section. 5.2.5) Finally, a list of earlier calls can be accessed by tapping on the "History" button (section 5.2.6).

5.2.3 Accessing contacts

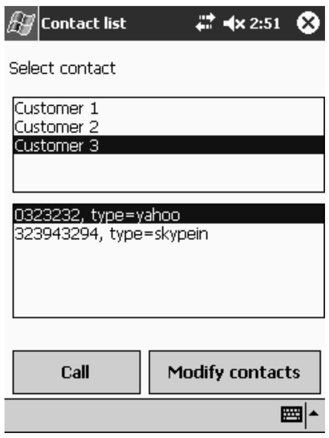
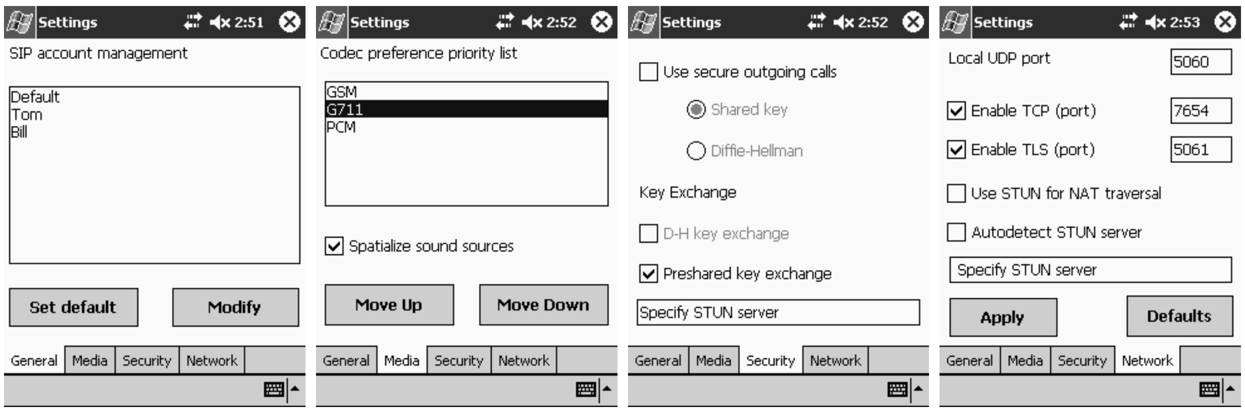


Figure 5.4: A contact list acts as a basic addressbook.

The contacts window consists of two context-sensitive lists (see figure 5.4). The top list groups the individual records in general categories (for example customers, colleagues, friends, etc.). The bottom list presents the records according to the general category chosen in the top list. The records themselves can be of different nature (e.g. landline/mobile phone numbers, SIP URIs, aliases, etc.), and can be assigned an arbitrary type (for example "work phone"), just as for the dektop GUI. The idea of two lists was chosen over the initial approach of having a single global list, because it enhances usability (since it is not necessary to search through the whole list everytime), while adding the convenience of grouping contacts into logical categories chosen by the user. A future extension could consider contacts exchange with other devices, taking advantage of interfaces for ad-hoc networking found in most of Pocket PCs (i.e. bluetooth, infrared, etc.).

5.2.4 Modifying settings

Minisip is an advanced SIP client, supporting a number of protocols for secure media sessions. As a result, there is a wide range of configurable parameters, which were impossible to fit on a single Pocket PC window without cramming the various options together in a way that violates our usability requirements. The solution was to organize the various settings into logical groups, then use a tab index control to assign a different tab to each group. In this way, the user is able to adjust only the settings of interest, without being presented with the entire settings list (see figure 5.5). In addition the settings windows required another level of interaction, since there are cases when a user can select only one choice from a set of available options (see for example the choice between diffie-hellman (DH) or shared key for secure calls in figure 5.5c), or is denied access to some of the options (the choice between DH and shared key exchange is not even possible if the user does not opt for secure outgoing calls). Fortunately, the properties of each interactive element[3] can easily be tweaked based on the user's actions (see section 5.3.4).



(a)
SIP account management
Default
Tom
Bill

(b)
Codec preference priority list
GSM
G711
PCM
☑ Spatialize sound sources

(c)
☐ Use secure outgoing calls
● Shared key
○ Diffie-Hellman
Key Exchange
☐ D-H key exchange
☑ Preshared key exchange
Specify STUN server

(d)
Local UDP port 5060
☑ Enable TCP (port) 7654
☑ Enable TLS (port) 5061
☐ Use STUN for NAT traversal
☐ Autodetect STUN server
Specify STUN server

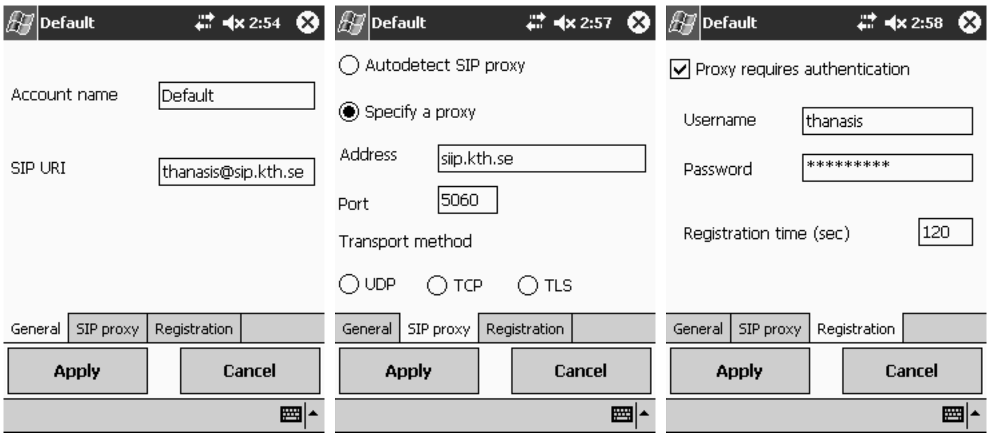
(a) (b) (c) (d)

Figure 5.5: Grouping settings together: (a) General (user profile configuration), (b) media, (c) security and (d) network. The user can tap on the respective tab on the bottom of the screen, and configure only the relevant information.

5.2.5 Managing user profiles

User profile management window follows the same guidelines as the settings window described previously (section 5.2.4). Minisip supports different user profiles, since a user might have accounts on multiple SIP proxies, or more than one users may use the same client (see figure 5.6).

[3]The definition of Microsoft's specific terminology - "controls" is given in section 5.3.4 - since this section concerns only the GUI design.


Default 2:54
Account name: Default
SIP URI: thanasis@sip.kth.se
General | SIP proxy | Registration
Apply | Cancel

Default 2:57
Autodetect SIP proxy
[✓ Specify a proxy]
Address: siip.kth.se
Port: 5060
Transport method
UDP TCP TLS
General | SIP proxy | Registration
Apply | Cancel

Default 2:58
[✓ Proxy requires authentication]
Username: thanasis
Password: *********
Registration time (sec): 120
General | SIP proxy | Registration
Apply | Cancel

(a) (b) (c)

Figure 5.6: Modifying information for an existing profile. This window is presented after selecting a profile from the general settings tab (see figure 5.5 a). The information are grouped into three categores: (a) General (name of the profile and SIP username), (b) SIP proxy settings and (c) proxy registration (authentication).

5.2.6 Call history window

This window is accessed from the dial window and offers a per-profile overview of the previous sessions (see figure 5.7). Note that this was not part of the original GUI, but it is useful for the end-users to be able to have a list of earlier calls. This is primarily of use to return calls or for a sequence of calls to the same party.

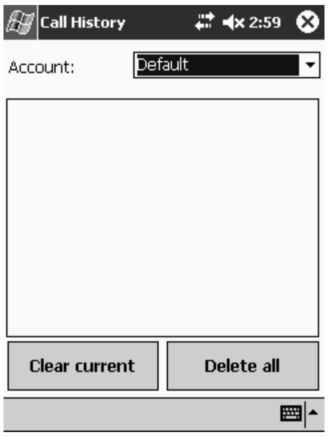

Call History 2:59
Account: Default
Clear current | Delete all

Figure 5.7: Implementation of a simple call log.

5.3 The porting process

5.3.1 Minisip software architecture

While the implementation of a Pocket PC native user interface was an important part of the porting proess, getting the source code to compile was of equal importance. Fortunately, there are provisions in the minisip source code for the Windows x86 platform (specifically the native thread model, windows sockets, windows system calls, etc.), since the port for that platform already exists. Being a subset of the Windows .NET Framework, a port to .NET CF benefits from the preexisting code for Windows x86 thus extensive modifications were not needed in order to compile for the Pocket PC environment. However, since various software components of minisip depends on third-party libraries, locating these libraries and porting them was done prior to compiling the source code of the application. These libraries are described in section 5.3.2.

Minisip itself consists of six distinct software components. The five libraries interoperate with the user interface, exchanging information according to the user's actions or as triggered by incoming SIP messages. Minisip uses a hierarchical software model. Libraries lower in the hierarchy are used by higher-level libraries. A description of each library, from lower to higher level follows:

- *libmutil*: Thread, semaphore, and mutex interface for Windows x86 and GNU/Linux platforms. Other utilities such as storage structures (including the circular buffer presented in chapter 3), are also implemented.
- *libmnetutil*: Network functions for Windows x86 and GNU/Linux (network socket setup, client-server model, canonical name resolution, etc.).
- *libmikey*: Implements the Multimedia Internet KEYing (MIKEY) protocol that exchanges session keys. The keys are used by the Secure RTP (SRTP) protocol for authenticated media sessions.
- *libmsip*: A SIP stack implementation.
- *libminisip*: Audio and video CODECs and drivers, RTP/SDP stack, media, and policy control.

The user interfaces are included in the *minisip* itself. GTKgui is the user interface based on GTK+ and was the basis for the design of the Pocket PC GUI. TextUI is the command line interface ported intact to the Pocket PC platform (see section 5.3.2). Finally there is also a version using the QT library, but it is currently outdated and non-functioning.

5.3.2 Additional libraries

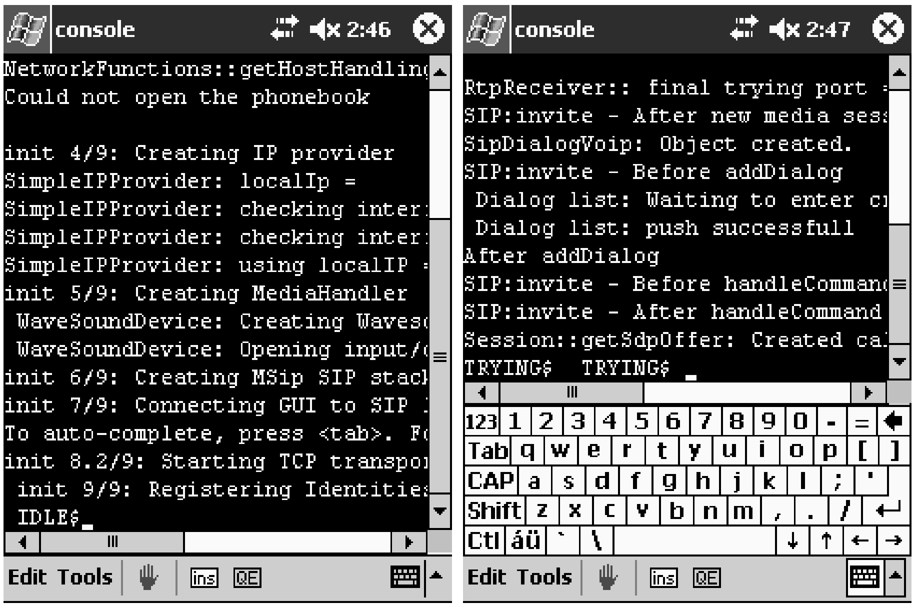
This section describes the third-party libraries that are used by minisip on the Pocket PC platform.

- OpenSSL [39]: The secure sockets layer library provides a set of cryptographic functions as well as an implementation of the SSL and TLS protocols. Among other uses, OpenSSL is the backend to SRTP and MIKEY implementations in minisip.
- udns [40]: Udns is a library to perform DNS queries in order to resolve IP addresses from canonical names. This is useful if for example a client wants to register to a SIP proxy, given the proxy's domain name. The client has to query the local DNS server for the IP address of the proxy. For GNU/Linux systems, minisip uses the *libresolv* library and for Microsoft's Windows the *WinDNS* application programming interface (API). Unfortunately, the WinDNS API is not available in .NET CF, so udns was ported and used. Currently there is an ongoing effort to adopt udns for all platforms.
- STL: The Standard Template Library is included in the C++ standard library, and is supported both on Windows x86 and GNU/Linux. It provides a set of storage utilities (containers) as well as the means to access them (iterators). In addition, it also features a set of sorting and searching algorithms. STL is used extensively in minisip. An example is the popular *vector* container (a dynamic array). However, STL is not part of .NET CF, but is supported by third-party implementations. Our initial approach was to use Giuseppe Govi's partial STL port [41], coupled with Dirk Jagdmann's *dojstream* library [42], for iostream support. These libraries were originally developed for the Pocket PC 2002 platform, and have not been updated since then. Our second approach was to use the more contemporary and complete *STLPort* library [43]. Both of these solutions worked, but STLPort was ultimately chosen, since it is frequently updated, and is more complete.
- PocketConsole [44]: This library provides a console device for Pocket PCs and was used for running minisip's command line user interface (textUI). Note that the Pocket PC environment uses the unicode character set system-wide [45]. However, minisip is written in ANSI C++, so in order for pocket console to output the characters to the device's screen properly, the original ANSI strings must be converted to unicode strings (see appendix B.3).

5.3.3 libmconsole: unicode text output and GUI debugging

The primary purpose of libmconsole is to accept ANSI strings (pointers to arrays of characters i.e. char *) and convert them to Pocket PC compatible unicode strings (pointers to w_char arrays).

Note that this not only aided the Pocket PC console output (see section 5.2.2), but also every part of the program that manipulates or compares strings. For examle, if libmconsole is not used, commands issued by the user (i.e. call, hangup, quit, etc.) are not recognized because Windows CE cannot compare ANSI strings. This library was designed to be cross-platform, in the sense that it performs ANSI to unicode conversions when compiling for an architecture that supports the ANSI character set (see figure 5.8).



```
NetworkFunctions::getHostHandlin
Could not open the phonebook

init 4/9: Creating IP provider
SimpleIPProvider: localIp =
SimpleIPProvider: checking inter
SimpleIPProvider: checking inter
SimpleIPProvider: using localIP =
init 5/9: Creating MediaHandler
 WaveSoundDevice: Creating Waves
 WaveSoundDevice: Opening input/
init 6/9: Creating MSip SIP stack
init 7/9: Connecting GUI to SIP
To auto-complete, press <tab>. F
init 8.2/9: Starting TCP transpo
 init 9/9: Registering Identitie
 IDLE$_
```

(a)

```
RtpReceiver:: final trying port =
SIP:invite - After new media ses
SipDialogVoip: Object created.
SIP:invite - Before addDialog
 Dialog list: Waiting to enter c
 Dialog list: push successfull
After addDialog
SIP:invite - Before handleComman
SIP:invite - After handleCommand
Session::getSdpOffer: Created ca
TRYING$  TRYING$ _
```

(b)

Figure 5.8: The command line user interface of minisip (textUI) running on PocketConsole with the help of libmconsole: (a) illustrates textUI's command prompt directly after initialization, (b) shows the user needs to use the soft input panel to enter commands and illustrates the output during SIP session establishment for a call.

Another issue with the Pocket PC environment is that it is difficult to switch between applications even though it supports multitasking. This was particularly a problem when trying to debug the GUI version, since it was impossible to simultaneously use the interface and monitor the debug messages on the device. Using libmconsole, debugging messages are encapsulated as the payload of UDP packets sent to a specified host. This host can capture the packets, extract and read the debug messages in real-time. To do so we used Wireshark [46], a widely available packet sniffer.

5.3.4 Interoperability between the GUI and the libraries

The GUI for minisip was implemented in Microsoft Visual C#[4] (MSVC#), a high level object-oriented language that is part of Microsoft's .NET CF. MSVC# instantiates window controls

[4]Pronounced C sharp.

(i.e. buttons, texboxes, lists, etc.) as objects. Consecutively, it is possible to tweak the object's properties, which was of particular use when programming the GUI (see requirements in sections 5.2.4 and 5.2.5). While the libraries were developed in ANSI C++, which is not part of the .NET CF, and were compiled as dynamically linked libraries (DLLs). Although MSVC# can acess functions contained in minisip DLLs, it will not recognize classes, and therefore any methods or variables that are class members. The inability to instantiate objects from minisip's libraries contradicts the requirement that class instances are mandatory in order for minisip to initialize as well as exchange messages between the libraries and the GUI. We approached the issue from two different perpectives:

The first approach was to compile the command-line interface (textUI) and the GUI into separate programs, then use messages to facilitate inter process communication (IPC) between them. The idea was that when a user issued a command from the GUI (i.e. pushed the call button), a message would notify the textUI process to execute the code corresponding to the equivalent command-line instruction. As proof-of-concept, a few commands (call, hangup, quit), were implemented. However, adopting this solution throughout the program would mean that a lot of code from the textUI had to be altered, in order to have complete functionality. In addition, some features such as user profiles would not be supported.

The second approach involved Microsoft's Common Language Runtime (CLR) virtual machine, which is also part of .NET CF. CLR can process code from every .NET language (also known as managed code), and translates it into native code at run-time. This means that the managed MSVC# code can instantiate objects from managed c++ (Visual c++ .NET) DLLs. However, since minisip code is unmanaged (because it was written in ANSI c++), special classes known as "wrappers" had to be implemented. Wrapper classes allow interoperability between managed and unmanaged code. Currently there is an ongoing effort to create wrappers for all libminisip classes, with a tool called SWIG [47]. This will allow for better utilization and integration of minisip DLLs from the GUI.

Chapter 6

Conclusions and Future Work

6.1 Playout buffering

This study proposed a mechanism for synchronized audio playback from SIP clients (the client used is minisip) on different network paths from the source to the device with different degrees of congestion. The purpose was to facilitate a natural flow in conversations, regardless of the nature and status of these network paths. In addition, such synchronization maintains the spatial effect of multiple sound streams.

The software infrastructure involves a client-server application, which monitors the end-to-end delay on individual network paths, then chooses the maximum of these values as the relative delay of the streams. Subsequenty, all participants adjust their audio playback to this artificial amount of delay, using an enhanced playout buffer implementation.

While the playout strategy as used in this study was based on keeping the streams synchronized, it remains a fact that minisip lacks a true adaptive buffer for dealing with jitter, packet loss, and end-to-end delay. Although currently there is no standardized playout scheduling scheme, there have been a number of different approaches to the problem, some of which can be considered for minisip. As is the case with our enhacements, the playout scheduling is done on the receiver of the audio streams, partly because no cooperation with the sender is required and partly due to network infrastructure independance. A basic distinction between adaptive playout bufferinf mechanisms concerns the timing of the buffer size adjustments.

6.1.1 Within talk-spurt (intra-talk-spurt)[1] adjustment algorithms

These algorithms constantly monitor the network and perform per packet playout scheduling of the audio stream. According to the varying delay characteristics of the network path, a packet can be assigned a longer or shorter playout time (for example, the 20ms audio data payload of a G.711 RTP packet, can be lenghtened or shortened to a value greater or shorter than 20ms). For this reason, algorithms of this category should be able to time-scale the audio waveform, thus the speaking rate, without affecting the perceived speech attribute (such as pitch, timbre, and voice quality). Waveform Similarity Overlapp-Add is a widely used algorith for speech compression or expansion [48]. Examples of intra-talksprut playout algorithms using WSOLA are [49] and [50].

6.1.2 Adjustments during silence periods (between talk-spurt algorithhms)

These algorithms work between talkspurts. Loss intolerant techniques avoid packet losses by creating as much end-to-end delay as it is required for smooth playout[2] [51]. Although simple to implement, these algorithms do not function well when the network jitter is high (because in this case the silence periods can be many and long). Other approaches are more loss tolerant, either by setting an upper limit for acceptable packet losses (in order to avoid large playout delays), or using a combination of metrics to keep the delay under a certain threshold[52] [53].

6.1.3 Adjustment algorithms based on perceived user experiencee (quality-based)

This category of algorithms started to emerge after the standardization of assesment methods aimed at evaluating the end-user perceived quality of audio communications. The telecommunications standardization sector of the International Telecommunication Union (ITU-T) has proposed a model for evaluating speech transmission quality in terms of end-user satisfaction (also known as the E-Model)[54]. The metric used in the E-model is known as transmission rating factor (R)[3]. Given the R factor value, the conversation can be categorised in respect to end user perceived quality [55]. Examples of algorithms using the E-model to schedule their playout strategy are [56] and [57].

[1]Talk-spurt, as used in this study, is the period of time during which audio is being transmitted. Speech consists of talk-sprut and silence periods.

[2]They work by estimating an average of this delay and setting it as the playout delay

[3]The equation and the parameters to compute R should be taken under account when implementing a quality-based buffer adjustment algorithm, but are out of scope of this thesis

6.1.4 Choosing an algorithm for adaptive playout buffering on minisip

Choosing an algorithm for implementing an adaptive buffering strategy is not straightforward, since there is no optimal solution as of yet.

- For low jitter (relatively stable network paths), a between talk-spurt algorithm is probably the best choice, given the added advantage of simplicity.
- For high jitter, intra-talk-spurt algorithms can compensate with the very fast adaptation (per-packet) to changing end-to-end delay. However in this case, the system has to be able to detect rapid delay changes accurately and timely (this is not part of the algorithm, but can be supported by a mechanism similar to syncd - see section 3.2). Complexity of time-scaling the audio from each incoming RTP packet's payload must also be taken under account.
- The novelty of qualitative based approaches, with the added advantage of standardized models for perceptive voice communication quality is also something to be considered. However, one must take into account that the current models (i.e. E-Model), require many assumptions in order to be used as the basis for quality-based algorithms (thus leaving out many possible scenarios).

6.2 The Pocket PC port for minisip

The effort for porting minisip to the Poccket PC platform, resulted in supporting two different user interfaces (a command-line and a GUI) as well as the complete set of features for minisip libraries. In addition it has produced a lot of documentation (see appendix B), thus creating the infrastructure for future developers to improve upon. Currently, the developers are focused on three different areas:

- Merging the PPC port with the trunk source.
- Implementing an autobuild system for building daily binaries.
- Using the wrapper functions in libminisip to connect the GUI with the libraries.

Future efforts could involve power management mechanisms, so that minisip goes to sleep mode whenever it is not used, therefore conserving power; enhancing the audio driver [58]; better integration with Windows CE (e.g. using popup windows during calls, icons on the taskbar, etc.); integration of the RTCP/NTP code from a previous effort [35]; and using Secure Digital card media for user authentication [59].

References

[1] "Sound Waves: A concise survey of the fundmental properties of sound waves (visited on 8th Jul., 2006)." [Online]. Available: http://www.du.edu/~jcalvert/waves/soundwav.htm

[2] B. Jens, *The Psychophysics of human sound localization.* MIT Press, 1997.

[3] Gestur Björn Christianson, "Information processing in the interaural time difference pathway of the barn owl," Ph.D. dissertation, California Institute of Technology, Pasadena, California, 2006. [Online]. Available: http://etd.caltech.edu/etd/available/etd-11212005-110457/unrestricted/GBCthesis.pdf

[4] "Primers Auditory Perception - Localization Primers: Auditory Perception - Localization (visited on 15th Jul., 2007)." [Online]. Available: http://www.ausim3d.com/about/AuWeb_perception.html

[5] S. Holland and D. Morse, "Audio GPS: Spatial Audio in a minimal attention interface," in *3rd International Workshop on HCI with Mobile Devices*, 2001.

[6] Thomas M. Drewes, Elizabeth D. Mynatt and Marybeth Gandy, "Sleuth: An audio experience," in *International Conference on Auditory Display.* Georgia Institute of Technology, April 2000.

[7] *The Virtual Reality Modeling Language, International Standard*, 14772nd ed., ISO/IEC, January 1997. [Online]. Available: http://tecfa.unige.ch/guides/vrml/vrml97/spec/

[8] Russell D. Shilling, Tomasz Letowski and Russell Storms, "Spatial auditory displays for use within attack rotary wing aircraft," in *International Conference on Auditory Display.* Georgia Institute of Technology, April 2000.

[9] Ignacio Sanchez Pardo, "Spatial audio for the mobile user," Master's thesis, Royal Institute of Technology (KTH), department of Microelectronics and Information Technology (IMIT), February 2005.

[10] S. H. Foster and E. M. Wenzel, "Virtual acoustic environments: The Convolvotron," *Computer Graphics*, vol. 4, p. 386, 1991.

[11] "DSP Boards directory (visited on 26th Jan, 2007)." [Online]. Available: http://www.dspguru.com/hw/boards.htm

[12] F. Rumsey, "Spatial audio and sensory evaluation techniques - context, history and aims," in *Spatial audio and sensory evaluation techniques.* Guildford, UK, April 2006.

[13] "Spatial audio at Multimedia Computing Group, Georgia Institute of Technology (visited on 10th Jul., 2006)." [Online]. Available: http://www.gvu.gatech.edu/gvu/multimedia/spatsound/spatsound.html

[14] "Aladdin Interactive DSP (visited on 26-Jan-2007)." [Online]. Available: http://www.hitech.se/development/products/aladdin.htm

[15] "LabView DSP (visited on 26th Jan. 2007)." [Online]. Available: http://www.ni.com/dsp/

[16] "SB-USB: Universal DSP JTAG Emulator with USB interface (visited on 26-Jan-2007)." [Online]. Available: http://www.domaintec.com/SBusb.html

[17] S.H. Kurniawan, A.J. Sporka, V. Nemec and P. Slavik, "Design and evaluation of computer-simulated spatial sound," in *Proceedings of the 2nd Cambridge Workshop on Universal Access and Assistive Technology (CWUAAT'04)*, 2004, pp. 137–146.

[18] V. Hardman and M. Iken, "Enhanced Reality Audio in Interactive Network Environments," in *Proceedings of the Enhanced Reality Audio in Interactive Networked Environments conference, Pisa, Italy*, December 1996.

[19] A. Ronald, M. Daily and J. Krozel, "Advanced Human-Computer Interfaces for Air Traffic Management and Simulation," in *Proceedings of 1996 AIAA Flight Simulation Technologies Conference*, July 1996.

[20] Jin-Young Yang, et al., "A design of a streaming system for interactive television broadcast," in *Circuits and Systems, (ISCAS).* Electronics Telecommunications Research Institute of Taejon, May 2000.

[21] Atzori, L. Lobina, M.L., "Playout buffering in ip telephony: a survey discussing problems and approaches," *IEEE Communications Surveys & Tutorials*, vol. 8, pp. 36–46, 3rd. Qtr. 2006.

[22] C. Perkins, O. Hodson and V. Hardman, "A survey of packet loss recovery techniques for streaming audio," *IEEE Network*, vol. 12, pp. 40–48, Sep/Oct 1998.

[23] J. Postel, *RFC791: Internet Protocol (visited on 26th Jan. 2007)*, Information Sciences Institute, University of Southern California, September 1981. [Online]. Available: http://tools.ietf.org/html/rfc791

[24] *RFC793: Transmission Control Protocol (visited on 26th Jan. 2007)*, Information Sciences Institute, University of Southern California, September 1981. [Online]. Available: http://tools.ietf.org/html/rfc793

[25] *RFC768: User Datagram Protocol (visited on2 26-Jan-2007)*, Information Sciences Institute, University of Southern California, August 1980. [Online]. Available: http://tools.ietf.org/html/rfc768

[26] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley and E. Schooler, *RFC3261: Session Initiation protocol (visited on 26th Jan. 2007)*, Internet Engineering Task Force (IETF), August 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3261.txt

[27] Pavlos Papageorgiou, "A Comparison of H.323 vs SIP." University of Maryland at College Park, USA, June 2001.

[28] T. Berson, "Skype security evaluation." Anagram Laboratories, October 2005.

[29] H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson, *RFC3550: RTP: A Transport Protocol for Real-Time Applications (visited on 26th Jan. 2007)*, Internet Engineering Task Force (IETF), July 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3550.txt

[30] H. Schulzrinne and S. Casner, *RFC3551: RTP Profile for Audio and Video Conferences with Minimal Control (visited on 26th Jan. 2007)*, Internet Engineering Task Force (IETF), July 2003. [Online]. Available: http://tools.ietf.org/html/rfc3551

[31] Oscar Scantillana, "RTP redirection using a handheld device with Minisip," Master's thesis, Royal Institute of Technology (KTH), department of Microelectronics and Information Technology (IMIT), February 2006.

[32] "SIP Express Router (visited on 26-Jan-2007)." [Online]. Available: http://www.iptel.org/ser/

[33] Erik Eliasson, "Minisip design overview," Royal Institute of Technology, Tech. Rep. TRITA-ICS/ECS R 06:04, ISSN-1653-7238, ISRN KTH/ICS/ECS/R-06/04-SE, 2006.

[34] "Why Free Software is better than Open Source (visited on 25th Jan, 2007)." [Online]. Available: http://www.fsf.org/licensing/essays/free-software-for-freedom.html

[35] Franz Mayer, "Adding NTP and RTP to a SIP User Agent," Master's thesis, Royal Institute of Technology (KTH), department of Microelectronics and Information Technology (IMIT), 2006.

[36] M. Baugher, D. McGrew, M. Naslund, E. Carrara and K. Norrman, *RFC3711: The Secure Real-time Transport Protocol (SRTP) (visited on 26th Jan. 2007)*, Internet Engineering Task Force (IETF), March 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3711.txt

[37] "The NistNet network emluator package (visited on 26-Jan-2007)." [Online]. Available: http://www-x.antd.nist.gov/nistnet/

[38] Mark Carson and Darrin Santay, "NIST Net, A Linux-based Network Emulation Tool," *ACN Computer Communication Review*, vol. 3, pp. 111–126, June 2003.

[39] "OpenSSL: A toolkit implementing SSL v2/v3 and TLS protocols with full-strength cryptography world-wide. (visited on 19th Feb., 2007)." [Online]. Available: http://www.openssl.org/

[40] "UDNS is a stub DNS resolver library with ability to perform both syncronous and asyncronous DNS queries. (visited on 19th Feb., 2007)." [Online]. Available: http://www.corpit.ru/mjt/udns.html

[41] "STL for eMbedded Visual C++ - Windows CE (visited on 26-Jan-2007)." [Online]. Available: http://www.syncdata.it/stlce/

[42] "dojstream: A C++ iostream compatible library (visited on 26-Jan-2007)." [Online]. Available: http://llg.cubic.org/tools/dojstream/

[43] "STLport: A multiplatform ANSI C++ Standard Library implementation. (visited on 26-Jan-2007)." [Online]. Available: http://www.stlport.org/product.html

[44] "PocketConsole is a software, which provides a console device for Pocket PCs. (visited on 19th Feb., 2007)." [Online]. Available: http://www.symbolictools.de/public/pocketconsole/index.html

[45] "Ten Tips for Programming for Microsoft Windows CE (visited on 26-Jan-2007)." [Online]. Available: http://msdn2.microsoft.com/en-us/library/ms834193.aspx

[46] "Wireshark is a network protocol analyzer (visited on 15th Jul., 2007)." [Online]. Available: http://www.wireshark.org/

[47] "SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. (visited on 15th Jul., 2007)." [Online]. Available: http://www.swig.org/

[48] Marc Roelands and Werner Verhelst, "Waveform Similarity Based Overlap-Add (WSOLA) for Time-Scale Modification of Speech: Structures and Evaluation," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Minneapolis, USA.* Vrije Universiteit Brussel, 1993.

[49] Yi J. Liang, Niko Färber and Bernd Girod, "Adaptive Playout Scheduling Using Time-Scale Modification in Packet Voice Communications," in *IEEE International Conference on acoustics, Speech, and Signal Processing.* Salt Lake City, UT, USA, 2001.

[50] Yi J. Liang et al, "Adaptive playout scheduling and loss concealment for voice communication over IP networks," *IEEE Transactions on Multimedia*, vol. 5, no. 4, pp. 532–543, December 2003.

[51] R. Ramjee, J. Kurose, D. Towsley and H. Schulzrinne, " Adaptive playout mechanisms for packetized audio applications in wide-area networks," *IEEE Conference on Computer Communications*, vol. 2, pp. 660–688, June 1994.

[52] Cormac J. Sreenan, Jyh-Cheng Chen, Prathima Agrawal and B. Narendran, "Delay Reduction Techniques for Playout Buffering," *IEEE Transactions on Multimedia*, vol. 2, no. 2, pp. 88–100, June 2000.

[53] K. Fujimoto, S. Ata and M. Murata, "Playout control for streaming applications by statistical delay analysis," *IEEE International Conference on Communications*, vol. 8, pp. 2337–2342, 2001.

[54] *The E-model, a computational model for use in transmission planning (visited on 16-Feb-2007)*, International Telecommunication Union (ITU), September 1999. [Online]. Available: http://www.itu.int/rec/T-REC-G.109-199909-I/en

[55] *Definition of categories of speech transmission quality (visited on 16-Feb-2007)*, International Telecommunication Union (ITU), March 2003. [Online]. Available: http://www.itu.int/rec/T-REC-G.107-200503-I/en

[56] L. Atzori and M. Lobina, "Speech Playout Buffering based on a Simplified Version of the ITU-T E-Model," *IEEE Signal Processing Letters*, vol. 11, no. 3, pp. 382–385, March 2004.

[57] K. Fujimoto, S. Ata and M. Murata, "Adaptive playout buffer algorithm for enhancing perceived quality of streaming applications," *IEEE Global Telecommunications Conference*, vol. 3, pp. 2451– 2457, November 2002.

[58] Ali Nesh-Nash, "Voice over IP in a resource constrained environment," Master's thesis, Royal Institute of Technology (KTH), department of Microelectronics and Information Technology (IMIT), June 2006.

[59] Pan Xuan, “Secure Voice over IP with Smart Cards,” Master’s thesis, Royal Institute of Technology (KTH), department of Microelectronics and Information Technology (IMIT), 2007.

Appendix A

Accurate timekeeping using NTP

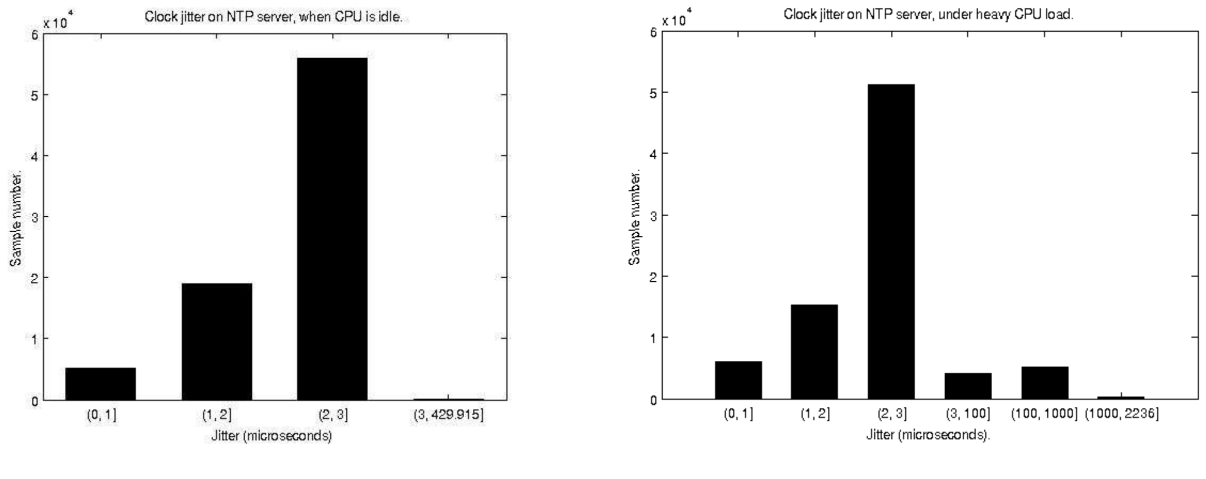
The synchronization mechanism presented on chapter 3, relies on the NTP protocol in order to synchronize all the peers in the conversation against a common time reference. In addition, syncd relies on NTP data in order to determine the end-to-end delay of a network link. This section addresses several issues concerning time accuracy when synchronizing to an NTP server, extracting NTP timestamps from local clocks, and propagating NTP packets throughout the network. It also provides some technical background on the tools used to conduct the accuracy measurements.

A.1 NTP internal clock jitter on the NTP server

NTP internal clock jitter refers to short-term random fluctuations in computer clock readings. The time needed to read the computer clock[1], depends on the load of the machine and the scheduling of the processes. For example, if another process has priority, the time scheduler of the kernel gives precedence to this process, keeping the other processes on hold. The process that was created in order extract the timestamp from the kernel clock has to wait before it is executed. This waiting time is called internal jitter, and can lead to inaccurate readings. Figure A1 illustrates the internal clock jitter of our router (a pentium 3 machine). When the CPU is idle (by idle we mean that there is no other program prominently consuming processing time and resources), 99% of samples are less or equal to 3 μ s. The rest 1% is no larger than 0.429 μ s, well below the 1ms precision of the measurements in chapter 4. When the CPU is stressed, a lot more interrupts have to be scheduled and assigned CPU time, thus the internal jitter tends to have larger and more dispersed values. In our example, 1% of the CPU click reads have values over 1 ms, the largest being 2.236 ms (thus a potential minisip client receiving a syncd packet from the server, could find calculate the end-to-end network delay with an inaccuracy of roughly 2ms). Other experiments with more CPU

[1]in UNIX systems this is done using the system call gettimeofday()

load indicated that internal clock jitter can be have values of the magnitude of tens of milliseconds.



(a) (b)

Figure A.1: Internal clock jitter of the computer used as router in chapter 4. Under nominal CPU load, jitter does not exceed our accuracy level (1ms), thus it practically does not affect our measurements (a). Under heavy CPU load, some values exceed the 1ms threshold, thus can affect syncd's evaluation of the network link's end-to-end delay (b).

Technical background: The program used in order to measure internal jitter can be found at the path ntp/jitter.c of the ntp suite installation directory[2]. The command *gcc jitter.c;./a.out ¿ jitterdata* generates a tracefile of nanoseconds called jitterdata, which can be imported into GNUplot, matlab, etc. for plotting. Simple shell scripts running infinite loops can be used In order to generate arbitrary amounts of CPU load. The scenario described above used a perl script that forked threads running infinite loops.

A.2 NTP external clock jitter

External network jitter refers to the propagation time of the network link that connects an NTP server with an NTP client. In our experiments, all clients reference the timestamps of the local NTP server in order to measure their link delays, thus precise timekeeping is not needed on the server. However real-world scenarios might require an efficient approximation of the current time, and it is up to the reader/user to select a suitable time server, both reliable, and relatively near[3] the local network of the experiment testbed. For example, the testbed that was used in the communications systems laboratory (domain wireless.it.kth.se), accessed a local NTP server, 3 hops away:

[2]NTP suite for unix systems can be downloaded from http://www.ntp.org/downloads.html (visited on Feb. 8, 2007)

[3]The terms that define distance (near, far) on a set of interconnected networks are closely related to the number of hops encountered in-between, rather than conveying the usual meaning of geographical distance.

```
traceroute to ntp.kth.se (130.237.32.95), 30 hops max, 40 byte packets
1  130.237.15.194  0.277 ms   0.246 ms   0.273 ms
2  kf4-ke5n-p2p.gw.kth.se (130.237.211.194)  0.242 ms   0.184 ms   0.272 ms
3  cn5-kf4-p2p.gw.kth.se (130.237.211.205)  0.602 ms   0.460 ms   0.521 ms
4  ea4-cn5-p2p.gw.kth.se (130.237.211.221)  0.729 ms   0.714 ms   0.668 ms
5  seiko.ite.kth.se (130.237.32.95)  0.603 ms   0.440 ms   0.400 ms
```

The average round-trip time for that server was less then 1 millisecond, thus external clock jitter did not affect the accuracy of our server's clock (round trip time was determined by pinging the host ntp.kth.se, average of 60 packets):

```
rtt min/avg/max/mdev = 0.441/0.590/5.378/0.624 ms
```

In order to get a sense of the importance of referencing a local NTP server, these are the results of the round trip time, when pinging ntp.kth.se (located in Sweden), from a Greek host:

```
62 packets transmitted, 62 packets received, 0\% packet loss
round-trip (ms)  min/avg/max = 74/74/75
```

A.3 Frequency fluctuations (drift)

Over time, a clock is prone to drift. This is particularly true to computer clocks, which are based on the readings from crystal oscillators. The metric we are using is frequency offset, which can be obtained from NTP statfiles:

```
[date] [time] [time offset] [freq. offset] [RMS jitter (seconds)]
[Allan deviation (PPM)] [Clock discipline time constant]

54110 1855.501 -0.000003000 3.113245 0.013243245 0.020392 10
54110 1921.500 -0.000202000 3.112344 0.023462674 0.019322 10
54110 1987.503 0.000025000 3.103214 0.028663561 0.013543 10
```

The frequency offset is actually the difference of the actual frequency minus the nominal frequency, which is always equal to 1 Hz. Figure A.2, illustrates the results of monitoring the frequency drift of the NTP clock of the router (see chapter 4). The experiments took place over the course of approximately two days, during which time, we gathered frequency data from the ntp statfiles. Unfortunately all the common computer clocks are not very accurate. Computer clocks are based on readings from quartz chrystal oscillators which are temperature dependent. A thermal stability in the room is important, in order for an oscillator to keep providing precise readings. Even an error[4] of only 0.001% would make a clock be off by almost one second per day. Frequency offset

[4]Error is defined here as the deviation from the correct time value.

is measured in PPM (parts per million). One PPM (Part Per Million) corresponds to 0.0001% (10^{-6}) error. Using this rule of thumb to tell how much time the clock lost over an elapsed period. Some examples of calculating the clock drift per day from ppm readings are illustrated below.
$3ppm = 3 * 10^{-6} = 0.000003 * 3600 * 24 seconds/day = 259.2 ms/day$
$2ppm = 2 * 10^{-6} = 0.000002 * 3600 * 24 seconds/day = 172.8 ms/day$

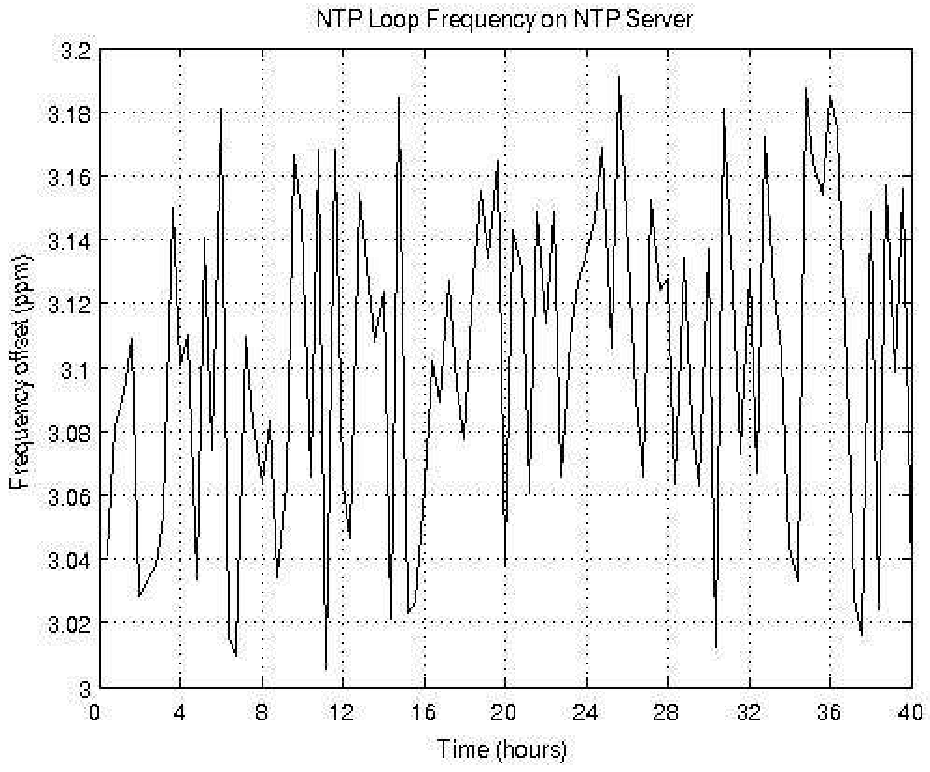


Figure A.2: Frequency offset for the clock of an IBM T21 (Pentium 3) computer.

Note that in the case of figure A2, the clock drifts were in the magnitude of tens of milliseconds per hour. The clock was relatively precise, because the experiment room maintained the same levels of temperature throughout the course of the experiment (so the oscillator provided stable readings). Also, when conducting the experiments in chapter 4, we synchronized with ntp.kth.se every 10 seconds, in order to correct any small clock drifts that might have occured.

Appendix B

Code listings

This appendix contains excerpts of code from software implemented during this study. For reasons of simplicity, we refrain from listing the complete code. Instead, we only list code reffered from various chapters of the thesis, providing pointers on how to obtain the complete sources.

B.1 Adding delay to minisip playout

The method for adding delay to the minisip buffer is calles setDelay. It accepts an integer argument representing a desirable delay value in milliseconds. SetDelay is a member of the CircularBuffer class contained in the libmutil utility library. It offsets the write index pointer by the amount corresponding to the delay argument given, and also fills the buffer space between the current and the offset value with previous audio data, in order to avoid gaps in audio playback..

```
void CircularBuffer::setDelay(int ms) {
    int doffset = ms;
    int lenLeft = 0;
    byteCounter += (unsigned long)ms;

    // If buffer is full, loop and start from the beginning
    if ( writeIdx + doffset > getMaxSize() ) {
        lenLeft = getMaxSize() - writeIdx; // size left until circular border crossing

        //Gapless playback: got to fill the buffer with something,
        //we might as well fill it with previous data
        memcpy(buf + writeIdx, (buf + writeIdx) - doffset, lenLeft * sizeof(short));
        memcpy(buf, (buf + writeIdx) - doffset, (doffset - lenLeft)* sizeof(short));

        writeIdx = doffset - lenLeft; // Start counting from the
                            // beginning of the buffer, write the rest of the data chunk.
    }
    // In this case, there are enough places in the buffer
   // to write our audio data chunk
    else {
       memcpy(buf + writeIdx, (buf + writeIdx) - doffset, doffset * sizeof(short));
        writeIdx += doffset;
        if (writeIdx >= getMaxSize())
            writeIdx -= getMaxSize();
    }

    size += doffset; // size: Current number of used elements from the buffer

}
```

B.2 syncd

Syncd is a client/server network application. Its operation is discussed in section 2.9. The client extracts timestamps from the kernel clock using the *gettimeofday()* system call (note that since our testbed was based on GNU/Linux machines, and the fact that this was a temporary solution until the RTCP code is available, we did not make the code portable to other architectures):

```
char* getTimestamp (){

struct timeval tv;
struct tm* ptm;
char time_string[40];
long milliseconds;

char *buffer;

gettimeofday (&tv, NULL);
ptm = localtime (&tv.tv_sec);
strftime (time_string, sizeof (time_string), "%Y-%m-%d %H:%M:%S", ptm);
milliseconds = tv.tv_usec;

buffer = (char *)malloc(sizeof (time_string) + sizeof (milliseconds));
sprintf(buffer, "%s.%06ld", time_string, milliseconds);

return buffer;
}
```

A pointer to the timestamp extracted from *getTimestamp()* is successively passed as an argument to *transmitUDPpacket()*. This function creates a socket, adds the payload and sends it to the designated IP address and port number (*payload* and *serverPort* respectively).

```
void transmitUDPpacket(char* servIP, char* payload, unsigned short port){
    int sock;  /* Socket descriptor */
    struct sockaddr_in serverAddress; /* Server address structure */
    int payloadLen;  /* payload length */

    payloadLen = strlen(payload);

    if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
        setErrorCode("socket() failed");

    memset(&serverAddress, 0, sizeof(serverAddress)); /* Initially set structure to zero */

    serverAddress.sin_family = AF_INET;                 /* Internet address family */
    serverAddress.sin_addr.s_addr = inet_addr(servIP);  /* Server IP address */
    serverAddress.sin_port   = htons(port);     /* Server port */

    if (sendto(sock, payload, payloadLen, 0, (struct sockaddr *) /*Send the UDP packet*/
               &serverAddress, sizeof(serverAddress)) != payloadLen)
        setErrorCode("sendto() sent a different number of bytes than expected");

    close(sock);

    sleep(1);
}
```

Also note the TR_PERIOD macro which marks the transmission period in seconds. It is used in a simple loop in the *main()* function that endlessly sends UDP packets.

```
#define TR_PERIOD 0.001  /* Transmission period, in seconds */
...
while( 1 ){
    transmitUDPpacket(servIP, getTimestamp(), port);
    sleep(TR_PERIOD);
}
```

The server is based on signals. In particular, a SIGIO signal is generated whenever new I/O can be completed on a socket (in our case when new data arrives at the socket). Upon packet arrival the *SIGIOHandler()* function is invoked after receiving a SIGIO. This function computes the difference between the timestamp included in the packet payload, and the timestamp of the local kernel clock. Successively, the difference is pushed into a list, later to be read by minisip's circular buffer.

```
void SIGIOHandler(int signalType)
{
    struct sockaddr_in clientAddress;
    unsigned int clientAddressLength;
    int receivedMessageSize;
    char echoBuffer[ECHOMAX];

    do
    {
         clientAddressLength = sizeof(clientAddress);

            if ((receivedMessageSize = recvfrom(sock, echoBuffer, ECHOMAX, 0,
                   (struct sockaddr *) &clientAddress, &clientAddressLength)) < 0){
               if (errno != EWOULDBLOCK)
                   setErrorCode("recvfrom() failed");
           }
           else{
        printf("[%s] Got %s\n ",inet_ntoa(clientAddress.sin_addr), echoBuffer);
        list[listSize] = compareTimestamps(echoBuffer);
        listSize++;
           }
           while (receivedMessageSize >= 0);
      }
}
```

The actual function to compute the difference is called *compareTimestamps()*. It uses *getTimestamp()* (identical function to the one of the client) to extract the local timestamp, then does the comparison, and returns the result in precision of microseconds.

```
int compareTimestamps(char *incoming){

char *local;
char inc[18];
char loc[18];
double f_src = 0.0;
double f_dst = 0.0;
float delta;

local = getTimestamp();

strcpy(inc, &incoming[17]);
strcpy(loc, &local[17]);

f_src = atof(loc);
f_dst = atof(inc);

delta = f_src - f_dst;
delta = delta * 1000000;

return (int)delta;
}
```

Finally, the function getListAverage() computes and returns the average from all the values currently in the list (as an estimation of the channel delay). In addition it deletes all the list contents, for new data to be written.

```
int getListAverage(){

int average;

if (listSize < 0)
return 0;

int sum = 0;
    int i = 0;

for (i=0; i<=listSize; i++) {
         sum = sum + list[i];
list[i] = 0;
}

average = sum / listSize;

return average;
}
```

B.3 libmconsole: unicode for Pocket PC

This section provides a step-by-step explanation of the libmconsole source code. Note that the header files have been omitted, since we are concentrated at the functionality of the code.

```
#include "wceconsole.h"

#define IP_ADDR "130.237.15.238"
```

The IP_ADDR macro holds the IP address of the host that captures the UDP debug messages (see section 5.3.3). As it will be indicated below, this only has meaning if the target architecture is Pocket PC.

```
static CRITICAL_SECTION consoleMutex;

//void consoleOut (string inputString, CRITICAL_SECTION consoleOUT){
#ifdef  LIBMCONSOLE_EXPORTS
   LIBMCONSOLE_API void consoleOut (char* inputString){
#else
   void consoleOut (char* inputString){
#endif

#ifdef _WIN32_WCE
```

```
    static int socketIsOpen = 0;
    static SOCKET s;
    static SOCKADDR_IN targetDevice;
    EnterCriticalSection(&consoleMutex);

    if (socketIsOpen == 0){

        #ifdef _WIN32_WCE
            InitializeCriticalSection(&consoleMutex); //put this where init. is done
        #endif
        socketIsOpen = 1;
        s = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP); //open socket

        if (s == INVALID_SOCKET) {
            int serror = WSAGetLastError();
            wprintf(L"wceconsole: socket error :%d\n", serror);
        }

        memset (&targetDevice, 0, sizeof(SOCKADDR_IN));

        targetDevice.sin_family = AF_INET;
        targetDevice.sin_port = htons(60);
        targetDevice.sin_addr.S_un.S_addr = inet_addr(IP_ADDR);
}

sendto(s, inputString, strlen(inputString), 0, (SOCKADDR *)
               &targetDevice, sizeof (SOCKADDR_IN));

#endif
```

ConsoleOut is the cross-platform method that is being used for printing strings. The LIBMCONSOLE_API macro is defined in wceconsole.h and is a directive to the compiler to export the method consoleOut from libmconsole.dll. The first thing that consoleOut does is to check whether the architecture we are compiling for is Pocket PC. This is done by checking if the macro _WIN32_WCE is set to 1 (when compiling for Pocket PC the preprocessor sets it automatically). If this is true, then the socket interface is set up (if running for the first time), and a UDP message with the current string as payload is sent to the IP_ADDR specified above. Also note that the operations of consoleOut are considered to be in a critical region: since many threads may be accessing the method simultaneously, priority must be given to one thread over the other, in order to avoid printing inconsistencies.

```
    const char *consoleOutTemp = inputString;

    #ifdef _WCE_ARCH
        wchar_t buffer [256];
        mbstowcs(buffer,  inputString, 256 );
        wprintf(L"%s", buffer);
    #elif defined _OTHER_ARCH
             // The structure is maintained for future architectures
    #else
         printf("%s", consoleOutTemp);
        break;
    #endif
```

This section of code concerns the actual output of the string given as argument to the standard output. If compiling for Pocket PC, first we do a multibyte (ANSI) to wide character(UNICODE) conversion of the string given to console out as argument, then print it to stdout, using the wprintf function. For any other architecture, we just print the ANSI string as it is, (other architectures being either the windows xp or linux versions (they both support the ANSI character set). There is also provision for future architectures which may want to print to their own character set.

```
consoleOutTemp = NULL;
LeaveCriticalSection(&consoleMutex);
Sleep(100); // arbitrary
}
```

The final section of code contains clean-ups and a sleep timer adding some delay (so that the IP_ADDR host does not get flooded by UDP packets.

B.4 Obtaining the source code

Interested parties can currently obtain the source code from minisip's subversion repository: svn://svn.minisip.org/minisip/branches/ppcgui. The ppcgui branch includes the C# GUI source files, as well as the modified source code of the minisip libraries not mentioned here. In addition, it includes developer's documentation about the changes made to minisip libraries in order to compile for the Pocket PC, as well as debugging and deploying tutorials to familiarize future developers with the Visual Studio IDE.

COS/CCS 2007-07

www.kth.se