Master Thesis

Implementation of a SIP User Agent for Mobile Devices

Institute of Broadband Communications (E388)
Faculty of Electrical Engineering
Vienna University of Technology

Under Supervision of o.Univ.Prof. Dr.-Ing. Harmen R. van As
and Proj.Ass. Dipl.-Ing. Klaus Umschaden

Samya Chahin

Vienna, 15. of October 2004

____

(Signature)

Abstract

The Session Initiation Protocol (SIP) is becoming more and more the main protocol for the initiation of two-way calls over IP Networks. It has been chosen by the 3GPP organization for Voice over IP (VoIP) and general multimedia two-way applications.
Furthermore, on UMTS and wireless networks the use of high quality audio coding schemes for communication purpose is possible. The goal of this master thesis is to implement a SIP User Agent for mobile phones. Wireless devices, such as mobile phones, laptop computers and personal digital assistants are gaining wide popularity. Their computing capabilities are growing quickly. These devices can be connected to wireless networks with increasing bandwidth. Moreover, the need for corresponding software is increasing greatly. The SIP protocol is used in this thesis to initialize calls and the Real-time Transport Protocol is used for the voice data exchange. In the future networks will develop, which enables the possibility to transfer bigger amounts of data. Mobile phones will get more powerful and the need to build corresponding powerful applications will grow too.

Preface

This master thesis has been written at the Institut of Broadband Communication, Vienna University of Technic, Austria. First of all, I would like to thank my advisor DI Klaus Umschaden for his support before and during the creation of this thesis, a lot of inspiring discussions and some constructive criticism. He deserves a large part of the laurels for this work, for his continuous patience and guidance.
I would like to express my gratitude to the Institut of Broadband Communication for giving me the opportunity to complete my studies and to thank all my friends and colleagues from the institutes at Vienna University of Technology. I'm really proud of having a chance to do my master thesis under Prof. Van As's supervise. He and DI Klaus Umschaden great guidance and help let me have the opportunity to enter this so challenging and interesting field and finish my work successfully at last.
Then, I would like to thank all my family, specially my husband, for staying always by my side.

List of Tables

List of Figures

Glossary

3GPP	3G Partnership Project
ADPCM	Adaptive Differential Pulse Code Modulation
AMS	Application Management Software
API	Application Program Interface
ATM	Asynchronous Transfer Mode
AVP	Audio Video Profile
AWT	Abstract Windowing Toolkit
CDC	Connected Device Configuration
CF	CompactFlash Cards
CLDC	Connected Limited Device Configuration
CORBA	Common Object Request Broker Architecture
CPU	Central PRocessing Unit
CSRC	Contributing source
CVM	C Virtual Machine
GCF	Generic Connection Framework
GPRS	General Packet Radio System
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hyper Text Transfer Protocol
IETF	Internet Engineering Task Force
I/O	Input/Output
IP	Internet Protocol
IrDA	Infrared Data Association
J2EE	Java 2 Enterprise Edition

J2SE	Java 2 Standard Edition
J2ME	Java 2 Micro Edition
JAD	Java Application Descriptor
JAIN	Java API for Integrated Networks
JAM	Java Application Manager
JAR	Java Archive
JMF	Java Media Framework
JNI	Java Native Interface
JRE	Java Runtime Environment
KVM	Kilo Virtual Machine
MID	Mobile Information Device
MIDP	Mobile Information Device Profile
MIME	Multipurpose Internet Mail Extensions
MMAPI	Mobile Media Application Programming Interface
MMUSIC	Multiparty Multimedia Session Control
NIST	National Institute of Standards and Technology
OTA	Over The Air Provisioning
PCM	Pulse Code Modulation
PCMCIA	Personal Computer Memory Card International Association Card
PDA	Personal Digital Assistant
PDU	Protocol Data Units
PSTN	Public Switched Telephone Networks
QoS	Quality of Service
RMI	Remote Method Invocation
RTCP	Real- time Transport Control Protocol

RTP	Real-time Transport Protocol
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SMS	Short Message Service
SMTP	Simple Mail Transfer Protocol
SSL/TLS	Secure Socket Layer /Transport Layer Security
SSRC	Synchronization source
TCP	Transmission Control Protocol
UAC	User Agent Client
UAS	User Agent Server
UDP	User Datagram Protocol
UI	User interface
UMTS	Universal Mobile Telecommunications Service
URI	Uniform Resource Identifier
VoIP	Voice over IP
WAP	Wireless Application Protocol
WLAN	Wireless Local Area Network
WML	Wireless Markup Language
WTK	Wireless Toolkit
XML	Extenible Markup Language

1 Introduction

Wireless devices, such as mobile phones, laptop computers and personal digital assistants are getting wide popularity. Their computing capabilities are growing quickly. These devices can be connected to wireless networks with increasing bandwidth. Moreover, the demand for corresponding software is increasing greatly. The combined use of these technologies on personal devices enables people to access their personal information as well as public resources anytime and anywhere. Especially, it enables the mobile multimedia entertainment and communication.
Providing flexible and programmable multimedia services at a lower cost to the end users in a more efficient way has been one of the main motivations behind transition from a traditional circuit switched network, Public Switched Telephone Networks (PSTN), to a packet based network such as IP network. Most of the efforts underway now are limited to wire-line network, however. As personal communication and ubiquitous access have become more prevalent, it is necessary to come up with solutions which can support multiple applications such as mobile IP-telephony, multimedia conferencing and other streaming applications over wireless IP networks. Supporting multimedia operation over wireless links has to consider several factors, such as mobile device, session signaling, wireless network and audio/video coding schemes.

Second, several new protocols will form the basis of the mobile multimedia infrastructure: Bluetooth for short-range connectivity of devices, Wireless Application Protocol (WAP) for access to information from cellular phones, Real-time Transport Protocol (RTP) for transport of audio and video streams over IP networks, Session Initiation Protocol (SIP), which is a text-based protocol similar to Hyper Text Transfer Protocol (HTTP) and Simple Mail Transfer Protocol (SMTP), for initiating interactive communication sessions between users that include voice, video or chat. With these protocols, multimedia sessions over wired wireless network can be signaled and managed very well.

The implementation of the SIP User Agent on mobile phones of this master thesis is just a sample to show the possibility that all the latest technology in this field together can be combined and provide a direction. As wireless network multimedia standards are developing, more efforts should be devoted to support these standards in future solutions, which will enable interoperability, and ensure the rapid adoption of new the technology.

This Thesis consists of six Chapters. Chapter 1 gives an introduction about the SIP and RTP protocol and the motivation for the work presented in this Thesis. It also discusses the reasons for choosing SIP as the signaling protocol. A short description of the main topics and contributions of this work is also included. Chapter 2 explaines the currently available mobile devices on the market and a comparison between them. A short description about mobile applications is included too. Chapter 3 covers the J2ME framework. The functionality is shown through simple examples. A comparison of the existing Java Virtual Machines is made and the main packages used for mobile phones as the Connected Limited Device Configuration and Mobile Information Device Profile are explained in more detail. This Chapter shows how all J2ME components work together, how network connections can be estabished and the lifecycle of MIDlets. Finally, the wireless toolkit will be introduced, because it is the development tool used in this Thesis. Chapter 4 is a very important part of this Thesis and covers the SIP protocol. It starts with the definition and shows call flow examples. The SIP messages, requests and responses, are explained. The SIP components are presented and the SDP is introduced. As an example of SIP User Agents currently available on the market, is shown with its

functionality. Finally, the available Java SIP packages for mobile devices are given, explaining the architecture of each with a comparison and the reason is mentioned for why choosing one of them for this Thesis. Chapter 5 gives an overview about the RTP protocol and why it is so important for SIP in the real-time Data exchange. The RTP header fields are explained in detail and how RTP fits in the architecture. Finally, the available Java packages for mobile multimedia are intoduced.
Chapter 6 includes the practical part of the Thesis, which is the prototype. Starting with the development steps used and the requirements needed to let this implementation possible. This Chapter showes and explains how all components work together. Own implemented parts are shown by sequence and class diagrams. A general class architecture is shown and a screenshot of the prototype illustrates how it is really working. How sampled voice data are possible to be played back through the Mobile Media API and how the UA was tested and call establishement is verified, is explained in detail in this Chapter.

1.1 Motivation

Today, there are many kinds of mobile phones from different manufacturers, which use different platforms. Network operators, device manufacturers and end users have different needs and expectations regarding to the applications running on them.
Although at the beginning Internet has been used only for researcg purpose, today's Internet has increased usability enabling different kinds of media transmission. A new range of communication services based on the Internet Protocol (IP) are now under use or deployment. A promising new communication service area comes with the so called Internet Telephony which together with User Mobility provides an interesting new communication paradigm. In this manner it is not strange to expect that in a couple of years IP networking will eventually replace the existing networking infrastructure even for mobile phones.
User Mobility is becoming more a requirement than an optional feature. The increasing variety of wireless devices offering IP connectivity such as Laptops, Personal Digital Assistances (PADs), handhelds and digital cellular phones make this mobile scenario already reality. User Mobility can be viewed from two distinct aspects. Personal mobility is user related in the sense that a user has access to network services based on user's personal identification and geographical position. Host mobility is movement oriented in the sense that when using wireless devices, a user can cross different network, without loose of the network connection. IP-based mobile networks offer a new communication paradigm with many communications service scenarios.
Considering the aggregate value that results from a work that implements and evaluates one of these standard solutions this master thesis has found its purpose.

1.2 Goal

The goal of this master thesis is to implement a prototype for a SIP User Agent for mobile phones. Evaluation of the current technologies for mobile devices and if mobile phones are powerful enough are a main part of this thesis. This SIP User Agent, as seen from the name, uses the SIP protocol to build up calls and the RTP for the voice data exchange. I will go through a deep analysis and design for RTP and after testing two different available Java SIP packages, I will decide to use one of them. The implementation will be explained in detail in chapter 7. The J2ME Framework is covered in a chapter alone and the functionality is shown through simple examples. As the RTP plays an important role in the voice Data exchange it is covered in a chapter separately.

2 General Overview of Mobile Devices

2.1 Mobile Devices

Mobile devices are devices which have not to be connected to a cable while working with them. There are different groups of mobile devices. The life cycle of mobile devices are very short. The optimal end device for all needs does not exist. The requirements for a mobile device depends on the functionalities it should fullfill. Then it will be clear to decide for a Personal Digital Assistant (PDA), because of its compact size and its relative big and high resolution display or a mobile phone which exists in huge numbers in the market.
Generally there exists four different kinds of mobile devices. Mobile phones, PDAs, SimPads and Laptops. Mobile phones are mainly for making calls. PDA is a organizer with or without a built in phone. SimPad is a small computer with a touchscreen and could be used to enter the Internet. Laptop is a mobile computer with nearly the same power as Personal Computers these days. Table 2.1 shows an overview about the different groups:

	Mobile Phones Target Group: Mass Market, Teenagers up to Bussiness People Availability: Wide spread Prices: ranges are from 100 to 1000 euro. Operating System (OS): is SymbianOS (earlier EPOC) Programming Languages: Java (Java 2 Micro Edition), C++, Wireless Markup Language (WML), i-mode (it is a packet-based information service that delivers information to mobile phones and enables the exchange of email from handsets.) CPU: slow 16-bit or 32-bit CPU Memory: mostly < 50kB, in special devices could be up to 500kB. Wireless Connection Types: Short Message Service (SMS), Global System for Mobile Communications (GSM), General Packet Radio System (GPRS), Infrared Data Association (IrDA), Bluetooth and Universal Mobile Telecommunications Service (UMTS) Extras Cameras, Organizer, Email client

	Personal Digital Assistant (PDA) Target Group: Bussiness People, Special Applications Availability: Medium spread. Prices: ranges are from 400,- to 1.000,- euro OS: WinCE or PalmOS Programming Languages: Java (J2ME), C++, C, VisualBasic, WML, Hypertext Markup Language (HTML) CPU: faster 32-bit CPU Memory: 32MB to 64MB Wireless Connection Types: SMS, GSM, GPRS, IrDA Bluetooth, UMTS, Wireless Local Area Network (WLAN) and with AddOn-Cards, Extras: Cameras, Organizer, Emailclient, big Colour Display, Touchscreen, Office Software, Audio/Video-Player, could be extended by CompactFlash (CF) Cards or Personal Computer Memory Card International Association (PCMCIA) Card
	SimPad Target Group: Privat and Bussiness People, Special Applications Availability: Low Spread Prices: from 1.000,- to 2.000,- euro OS: WinCEOperating Programming Languages: Java (J2ME), C++, C, VisualBasic, HTML CPU: faster 32-bit or 64-bit CPU Memory: arround 32MB for Application Memory Wireless Connection Types: SMS, GSM, GPRS,IrDA, Bluetooth, UMTS, extended through WLAN and PCMCIA-Cards, Extras: Cameras, Organizer, Emailclient, Colour Display, Touchscreen, Office Software, Audio/Video-Player
	Laptop Target Group: Busssiness People, Field Staff Availability: Wide Spread Prices: from 1.000,- to 3.000,- euro OS: for example WinXP or Linux Programming Languages: Java (Java 2 Standard Edition), C++, C, VisualBasic, HTML CPU: faster 32-bit or 64-bit CPU Memory: arround 32MB up to many hundreds MB Wireles Connection Types: SMS, GSM, GPRS, IrDA, Bluetooth, UMTS, extended through WLAN and PCMCIA-Cards Extras: Cameras, Organizer, Emailclient, Colour Display, Touchscreen, Office-Software, Audio/Video-Player

Table 2.1 Properties of different mobile devices

Mobile phones are getting more and more powerful and take more and more the role of PDAs. Those phones are an interesting group for mobile applications because they are powerful, small enough for the daily use and there exists a huge number of them on the market. Probably those mobile phones will get more powerful in the next years with the same small size and will be the daily digital assistent for communication and information exchange, organization, comfort, security and remote control on the market and will replace the PDAs completely. Because Mobile phones will play such an important role in the future, I chose the focus of my Master Thesis to be about that group of devices.

The subsequent Table 2.2 shows advantages and disadvantages of the different devices.

Device	**Advantages**	**Disadvantages**
Mobile Phone	+ Small + Cheap + Wide Spread + Phone and programmmable device as one unit	- Small Display - Low Memory - Slow CPU - Complex Operation
PDA	+ Big display + Easy operation through touchscreen + Powerful + MDA: phone and programmmable device as one unit	- PDA: phone function only with extended modules - As mobile phone too big and too heavy - High price
SimPad	+ Very big display + Easy use through touchscreen + Powerful	- Too heavy and to big for a real mobile device - High price
Laptop	+ Very big display + Easy use through keyboard and mousepad + Powerful as a desktop personal computer	- Too heavy and too big for a real mobile device - High price

Table 2.2 Advantages and Disadvantages of the different mobile devices

2.2 Mobile Applications

„Mobile Applications“ or often also called „Wireless Applications“ are developed mainly to be used for mobile devices. Mobile devices could be programmable mobile phones, smartphones, PDAs or mobile small computer like SimPads and laptops. The mobile devices are small, very flexible and are location independent-that is why they are „Mobile“-and have either no or a wireless connection to other devices-that is why they are „Wireless“).
The mobile device together with the suitable application is called „Mobile Solution“or „Wireless Solution“. This is a system with very flexible and location independent properties and possibilities.
The trend is no longer to develop applications for laptops or PDAs but to have nearly every mobile phone together with special applications to have a multifunction device which opens wide opportunities. Mobile phones are small, cheap and widely spread. Most of the new mobile phones support the J2ME program interface.

Some years ago the company Palm released the first programmable PDA. Many developers around the world started to write applications for those devices which enabled the success of PDAs. Every end user was able to extend his PDA with the needed functionality.
Nowadays, there exist a huge number of applications for PDAs like Palm and PocketPC. After J2ME came out as a platform independent programming language for mobile phones and other mobile devices, they will have the same future. There exists already more than thousands of games and tools which can be downloaded to the mobile phones.

3 Java 2 Micro Edition

Java 2 Micro Edition (J2ME) is a Java based framework for mobile devices such as cellular phones. In this framework the following properties have been taken into account:

- Slow processor
- Little memory
- A small key pad
- Small display
- A limited source of power supply
- Low transfer speed compared to computers

The J2ME architecture defines configurations, profiles and optional packages as elements for building complete Java runtime environments that meet the requirements for a broad range of devices and target markets. Each combination is optimized for the memory, processing power, and I/O capabilities of a related category of devices.
Figure 3.1 gives an overview about the different Java frameworks and the Java Virtual Machines (JVM) in use.

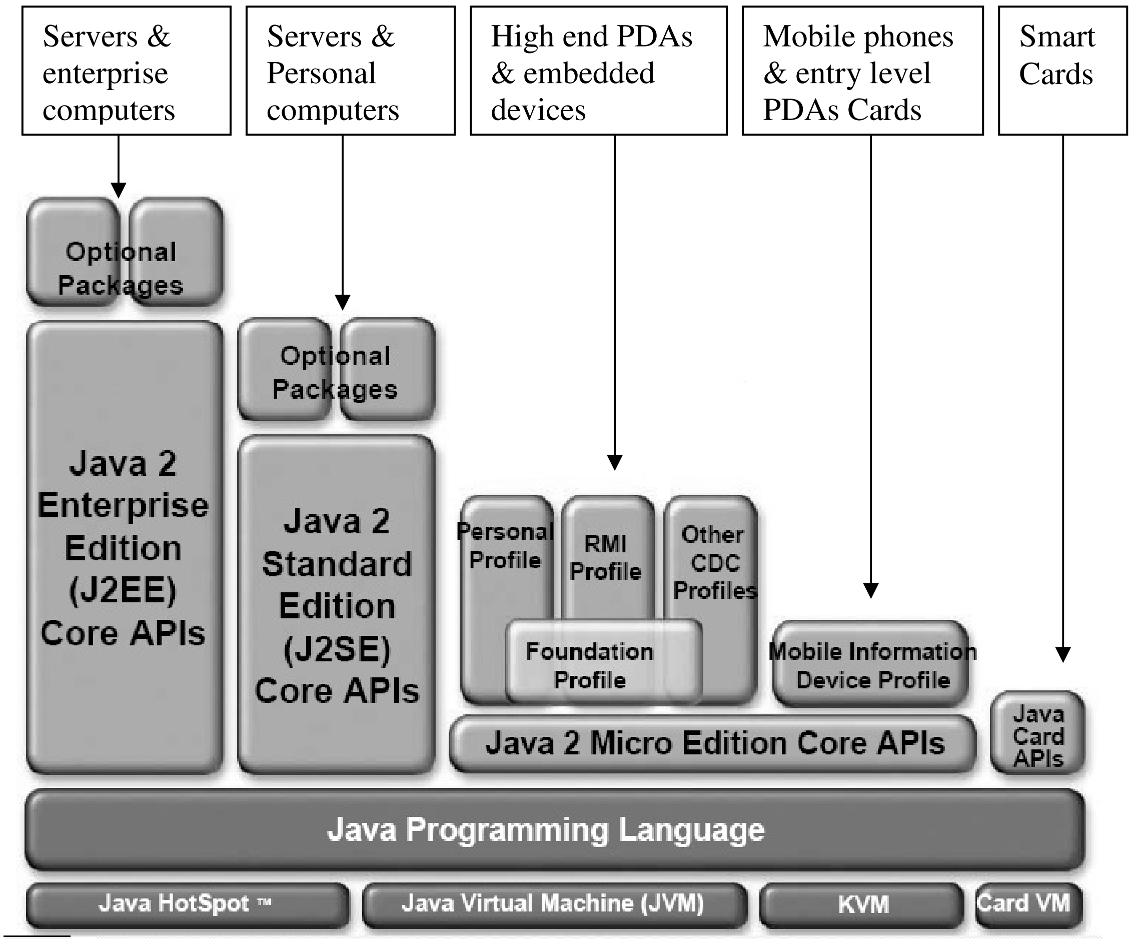

Servers & enterprise computers
Servers & Personal computers
High end PDAs & embedded devices
Mobile phones & entry level PDAs Cards
Smart Cards
Optional Packages
Optional Packages
Java 2 Enterprise Edition (J2EE) Core APIs
Java 2 Standard Edition (J2SE) Core APIs
Personal Profile
RMI Profile
Other CDC Profiles
Foundation Profile
Mobile Information Device Profile
Java 2 Micro Edition Core APIs
Java Card APIs
Java Programming Language
Java HotSpot ™
Java Virtual Machine (JVM)
KVM
Card VM

Figure 3.1: Java frameworks and related VM **[SUN]**

As seen from the Fihure 3.1 there exists also two other Java based frameworks which are Java 2 Standard Edition (J2SE) and Java 2 Enterprise Edition (J2EE).
J2SE is a Java-based, runtime platform that provides many features for developing Web-based Java applications, including database access, CORBA interface technology, and security for both local network and Internet use. J2SE is the core Java technology platform.
J2EE is a Java-based, runtime platform used for developing, deploying, and managing multi-tier server-centric applications on an enterprise-wide scale. J2EE builds on the features of J2SE and adds distributed communication, threading control, scalable architecture, and transaction management.

3.1 Architecture

All J2ME devices do not have the same perfomance capability or restrictions and at the same time J2ME should be valid for a wide range of devices. Therefore, J2ME is devided into two main parts, the configuration and the profile as shown in Figure 3.2.
The configuration describes the general, minmal functions for general device classes such as mobile phones. The configuration describes the suitable Virtual Machine and the suitable Application Program Interface (API). For each configuration there exist different profiles, which help to gain a complete Java Runtime Environment (JRE). The profile consists of a limit set of APIs, addressing only functional areas that were considered absolute requirements to achieve broad portability and successful deployments. The JVM layer is an implementation of a Java Virtual Machine that is customized for a particular device's host operating system and supports a particular J2ME configuration. There exist two type of JVM which are C Virtual Machine (CVM) and Kilo Virtual Machine (KVM) and will be explained in detail in the subsequent section. A MIDlet is a Java program for embedded devices, more specifically the J2ME virtual machine. Generally, these are games and applications that run on a cell phone. **[J2MICRO]**

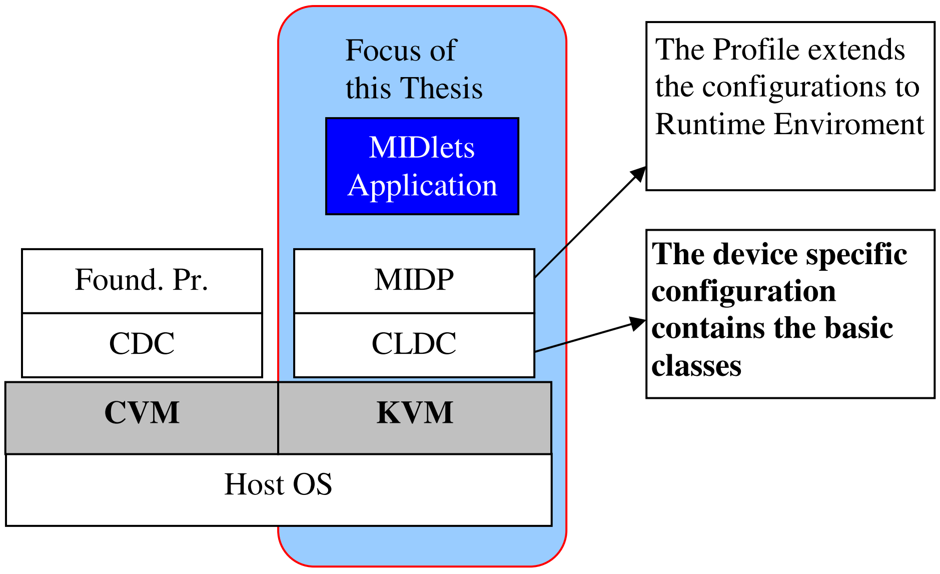

Focus of this Thesis
MIDlets Application
Found. Pr.
CDC
MIDP
CLDC
CVM
KVM
Host OS
The Profile extends the configurations to Runtime Enviroment
The device specific configuration contains the basic classes

Figure 3.2 J2ME with configuration, profile and MIDlets

In this Master Thesis I will concentrate mainly on the configuration for mobile phones and the suitable profile which is called Mobile Information Device Profile (MIDP). Other available configurations are just mentioned for completion.

Currently, there are two different types of configurations available: Connected Device Configuration (CDCl) and Connected Limited Device Configuration (CLDC), which is explained in detail in section 3.4.

3.2 Vitual Machine Layer

The JVM layer is an implementation of a JVM that is customized for a particular device's host OS and supports a particular J2ME configuration.
There exist two famous Virtual Machines in the environment of J2ME. They are KVM and CVM. These are the names of Java Virtual Machines for the CLDC (KVM) and the CDC (CVM). Those are designed specifically to work in the constrained environment of a handheld or embedded device. Additionally, they can be ported to different platforms. The CLDC and CDC specifications do not require the use of the KVM or the CVM, only the use of a JVM that fullfills the requirements of the CLDC or CDC specification.

3.2.1 C Virtual Machine

CVM is a full-featured JVM designed for devices needing the functionality of the Java 2 Virtual Machine feature set, but with a smaller footprint. This means less methods and less functionality. It is used by the CDC which is the goal of embedded devices. The implementation is not based on CVM, therefore I will not go into more detail about it.

3.2.2 Kilo Virtual Machine

The KVM, also known as the Kilo Virtual Machine, is a JVM designed for small, resource-constrained devices such as cellular phones, pagers, personal organizers and mobile Internet devices.
The goal of the KVM was to create the smallest possible complete JVM that can handle nearly all the aspects of the Java programming language, and that would also run in a resource-constrained device with only a few tens or hundreds of kilobytes of available memory. Therefore it is called Kilo Virtual Machine, which means just several kilobytes.
The KVM is implemented in the C programming language, so it can easily be ported onto various platforms for which a C compiler is available.
It is possible to start the KVM from the command line. Alternatively, on devices with user interface capable of launching native applications (such as Palm OS), the KVM can be configured to run in that way.
For devices that do not have such a user interface, the KVM provides a reference implementation of a facility called the Java Application Manager (JAM), which serves as an interface between the host OS and the virtual machine. The JAM assumes that applications are available for downloading as Java Archive (JAR) files by using a network or storage protocol (typically HTTP) implemented using the Generic Connection Framework (GCF). The JAM reads the contents of the JAR file and an associated descriptor file from the Internet, and launches the KVM with the main class as a parameter. For development and testing purposes, desktop implementations of the KVM can be configured to use the JAM as an alternative startup strategy.

The KVM does not support the Java Native Interface (JNI). Rather, any native code called from the virtual machine must be linked directly into the JVM at compile time. Invoking native methods is accomplished via native method lookup tables, which must be created during the build process [J2ME].

3.3 Configuration Layer

3.3.1 Connected Limited Device Configuration

The Connected Limited Device Configuration (CLDC) has the goal to offer a standard, small Java platform for resource constrained connected devices and is the used configuration for mobile phones and similar devices. Generally, they have the following characteristics:

- 16 or 32 Bit-CPU
- 160 kB to 512 kB of total memory budget available for the Java platform
- Connectivity to a network, often with a wireless connection and limited bandwidth (often 9600 bps or less)
- Low power consumption, often battery power is used

The CLDC defines the minimum required complement of Java technology components and libraries for small connected devices. Primary topics focused by this specification are:

- Java language and virtual machine features
- Core Java libraries (java.lang.*, java.util.*)
- Input/output
- Networking
- Security
- Internationalization

This CLDC Specification shall not address the following features:

- Application life-cycle management (application installation, launching, deletion)
- User interface functionality
- Event handling
- High-level application model (the interaction between the user and the application)

Those mentioned features are the job of the profile MIDP implemented on top of the CLDC. Future CLDC versions might include other areas. To restrict the scope of CLDC in order not to exceed the strict memory limitations it could be also possible later to exclude any particular device category**[RIGGS].**

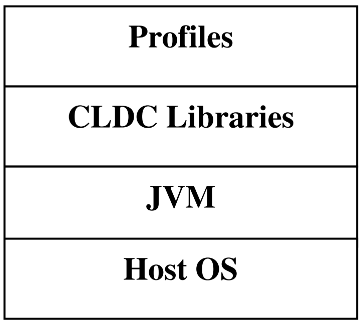


Figure 3.3 General J2ME Architecture

The general architecture of a typical CLDC device is shown in Figure 3.3. The heart of a CLDC implementation is the Java Virtual Machine. The virtual machine typically runs on top of a host OS. The JVM for this configuration is the KVM designed to use few memory suitable for limited devices. The „K“ from KVM stands for Kilo and expresses that few memory is used. On top of the virtual machine resides the Java libraries. These libraries are divided into two categories, those defined by the CLDC and those defined by profiles.

As mentioned the CLDC includes Java classes and Java applications are possible to be written with this configuration and as all Java applications a single entry point for the application should exists which is the main function used by the Virtual Machine:

public static void main(String[] args)

3.3.1.1 Restrictions of CLDC

In the CLDC there are the following restrictions:

1. **No floating point support:** The KVM doesn't support floating points. Floating point support was removed because the most of CLDC target devices do not have hardware floating point support, and since the cost of supporting floating point in software was considered too high. This is valid for the CLDC version 1.0. In version 1.1 floating points are included, because cellular phones are getting more powerful but few or even still no mobiles have this version preinstalled. For this Master Thesis version 1.0 was used.

2. **No Java Native Interface (JNI):** It is not possible to use libraries from other programming languages. A JVM supporting CLDC does not implement the JNI. The way in which the virtual machine invokes native functionality is implementation dependent. Support for JNI was eliminated mainly because of two reasons:

   1) the limited security model provided by CLDC assumes that the set of native functions must be closed .
   2) the full implementation of JNI was considered too expensive given the strict memory constraints of CLDC target devices.

3. **No user defined class loader:** The application can not influence the order the classees could be loaded. The JVM can only load the classes in a specific order.
4. **No reflections:** Reflection means allowing a Java program to inspect the number and the contents of classes, objects, methods, fields, threads, execution stacks and other

runtime structures inside the virtual machine. Consequently, a JVM supporting CLDC also does not support Remote Method Invocation (RMI), object serialization, or any other advanced features of J2SE that depends on the presence of reflective capabilities.

5. **No thread-groups or dameon-threads:** Only simple threads are allowed

6. **No serialisation of objects**

7. **No finalization method:** Before an object is garbage collected, the Java runtime system gives the object a chance to clean up after itself. This step is known as finalization and is achieved through a call to the object's *finalize* method.

8. **Weak references:** A JVM supporting CLDC does not support weak references. A weak reference is one that does not prevent the referenced object from being garbage collected and is a reference that does not keep the object it refers to alive. It is not counted as a reference in garbage collection. If the object is not referred to elsewhere as well, it will be garbage collected**.**

3.3.1.2 Class file verification

The VM goes through a process known as byte-code verification whenever it loads an untrusted class. This process ensures that the byte codes of a class are all valid; that the code never underflows or overflows the VM stack; that local variables are not used before they are initialized; that field, method, and class access control modifiers are respected; and so on. The verification step is designed to prevent the VM from executing byte codes that might crash it or put it into an undefined and untested state where it might be vulnerable to other attacks by malicious code. Byte-code verification is a defense against malicious hand-crafted Java byte codes and untrusted Java compilers that might output invalid byte codes**[OR]**.
A more compact and efficient verification solution has been specified for a typical CLDC target device because the static and dynamic memory footprint of the standard Java classfile verifier was too excessive.
The new classfile verifier operates in two phases, pre-verification and in-device verification. Pre-verification generally takes place off-device, e.g., on a server machine from which Java applications are being downloaded, or on the development workstation where new applications are being developed. In-device verification is carried out inside the device containing the virtual machine. The in-device verifier utilizes the information generated by the pre-verification tool.

3.3.1.3 CLDC Libraries

In CLDC, some classes are taken from the J2SE. Those resides in the packages *java.io*, *java.lang* and *java.util*. Due to the CLDC restrictions, those classes were changed or parts have been taken out. Additionaly, there exists classes in the Generic Connection Framework (GCF) as shown in Figure 3.4. This is a collection of classes with which connections as sockets or HTTP-requests could be implemented in an abstract manner. In J2SE, some of the classes are in the *java.io*- and *java.net*-Packages. The GCF does not implement any protocol. The protocols are defined either at profile level or in the end device. Consequently, it is possible that some mobile phones do not support all of the protocols.

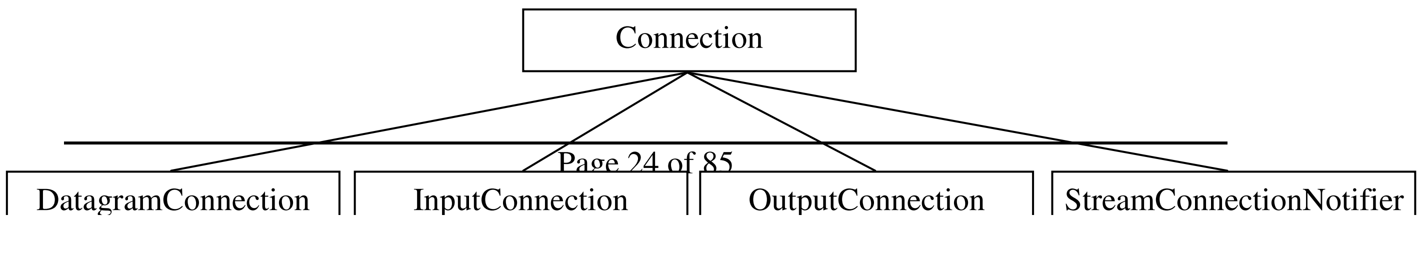


Figure 3.4 The CLDC Generic Connection Framework (GCF)

In Table 3.1, I illustrate the classes included in the CLDC. It consists of the following packages: *java.io*, which contains the classes responsible for input and output mainly through data streams, *java.lang* a subset of the J2SE package which holds the basic classes for the Java programming language, the *java.util* package, which holds some useful classes for the collection frameworks and the date and time conversion functions. Finally, the *javax.microedition.io* contains the classes for the GCF responsible for the network connections.

System classes (*java.lang*)	*Object*, *Class*, *Runtime*, *System*, *Thread*, *Runnable* (interface), *String*, *StringBuffer*, *Throwable*
Data type classes (*java.lang*)	*Boolean*, *Byte*,.*Short*, *Integer*, *Long*, *Character*
Collection classes (*java.util*)	*Vector*, *Stack*,*Hashtable*,*Enumeration* (interface)
Input/ Output classes (*java.io*)	*InputStream*, *OutputStream*, *ByteArrayInputStream*, *ByteArrayOutputStream*, *DataInput*(interface), *DataOutput* (interface), *DataInputStream*, *DataOutputStream*, *Reader*, *Writer*, *InputStreamReader*, *OutputStreamWriter*, *PrintStream*
Calender and Time classes (*java.util*)	*Calendar*, *Date*, *TimeZone*
Utility classes (*java.util* and *java.lang*)	*Random*, *Math*

Exception classes (*java.lang, java.io and java.util*)	*Exception*, *ClassNotFoundException*, *IllegalAccessException*, *InstantiationException*, *InterruptedException*, *RuntimeException*, *ArithmeticException*, *ArrayStoreException*, *ClassCastException*, *IllegalArgumentException*, *IllegalThreadStateException*, *NumberFormatException*, *IllegalMonitorStateException*, *IndexOutOfBoundsException*, *ArrayIndexOutOfBoundsException*, *StringIndexOutOfBoundsException*, *NegativeArraySizeException*, *NullPointerException*, *SecurityException*, *EmptyStackException*, *NoSuchElementException*, *EOFException*, *IOException*, *InterruptedIOException*, *UnsupportedEncodingException*, *UTFDataFormatException*
Error classes (*java.lang*)	*Error, VirtualMachineError, OutOfMemoryError*
GCF classes and interfaces (*javax.microedition.io*)	*Connection*, *ConnectionNotFoundException*, *Connector*, *ContentConnection*, *Datagram*, *DatagramConnection*, *InputConnection*, *OutputConnection*, *StreamConnection*, *StreamConnectionNotifierConnection*, *DatagramConnectio*, *InputConnection*, *OutputConnectionStream*, *ConnectionNotifier*

Table 3.1 Classes contained in the CLDC

3.3.2 Connected Device Configuration

CDC is the Connected Device Configuration and the typical configuration for PDAs and other devices with the following characteristic:

• 32 oder 64 bit-CPU
• At least 2 MB memory for Java
• Access to a network

The CDC contains nearly all of the J2SE classes and works with the normal JVM. Therefore, the CDC contains all functionalities, classes and interfaces of CLDC. There is only one profile available for CDC, which is called Foundation Profile. In this thesis I will not go into detail about this configuration, because for the current generation of mobile phones the CDC plays a minor role.

3.4 Profile Layer

3.4.1 Mobile Information Device Profile

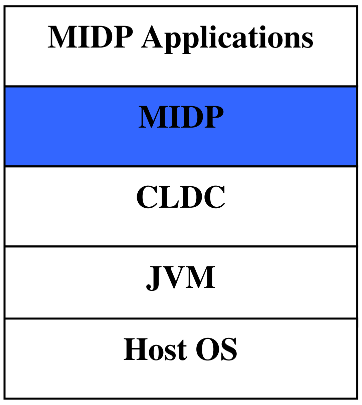


Figure 3.5 High level architecture of MIDP

The MIDP is the Mobile Information Device Profile and is built to operate on top of the CLDC. Usually, this profile is used by cellular phones, two-way pagers, and wireless-enabled personal digital assistants (PDAs) **[MIDP]**. It consists of a limit set of APIs, addressing only functional areas that were considered absolute requirements to achieve broad portability and successful deployments. These include:

- Application delivery
- Application lifecycle (semantics of a MIDP application and how it is controlled)
- Application signing model and privileged domains security model
- End-to-end transactional security (HTTP Security)
- MIDlet push registration (server push model)
- Networking
- Persistent storage
- Sound
- Timers
- User interface (UI) (including display and input, as well as the unique requirements for games).

Following aspects are outside the scope of MIDP:

- System-level APIs:
  The goal for MIDP APIs is to enable application programming, rather than enabling system programming. Thus, low-level APIs are beyond the scope of this specification. Therefore, power management or voice codecs are not included in MIDP APIs.

- Low-level security:
  The MIDP specifies no additional low-level security features other than those provided by the CLDC.

The Profile requires the following charcteristics in the end devices:

- The KVM must run in its own thread in the operating System in the end device
- Persistant memory access to access the network interface of the wireless.
- The MIDlet lifecycle (installation, choice, start, close and delete) must be managed by an application management software in the device. The exact procedure is not mentioned in the MIDP.

More charcteristics are shown in Table 3.2.

Display	Screen-size: 96x54, Display depth: 1-bit, Pixel shape (aspect ratio): approximately 1:1
Input	One or more of the following user-input mechanisms: one-handed keyboard, two-handed keyboard, or touch screen
Memory	256 kilobytes of non-volatile memory for the MIDP implementation and 8 kilobytes of non-volatile memory for application-created persistent data, 128 kilobytes of volatile memory for the Java runtime (e.g., the Java heap)
Networking	Two-way, wireless network support with limited bandwidth
Sound	The ability to play tones, either via dedicated hardware, or via software algorithm.

Table 3.2 MIDP Profile Characteristic

3.4.2 MIDlets and Lifecycle of MIDlets

A J2ME Application is called MIDlet, if it is build upon the MIDP. Like applets, MIDlets are managed in an execution environment that is slightly different from that of a Java application. The initial entry point to a MIDlet is not the *main* method, and the MIDlet is not allowed to tear down the JVM.

One way to easily build a Midlet is to use the Ktoolbar which is included in the WirelessToolkit of SUN Microsystems. It builds the Midlet automatically instead of writing all parts from scratch. In the practical part of my Master Thesis, I have used the WirelessToolkit which I will explain in more detail later.

The MIDlet must extend the *javax.microedition.midlet.MIDlet* class. The MIDP platform involves methods of the MIDlet to control the MIDlet's lifecycles. Additionally, MIDlet itself can request a change in its state. Listing 3.1 shows how a Midlet looks like.

```
public class MyMIDlet extends MIDlet {

         // Optional constructor
        MyMIDlet( ) {
         }

        protected void startApp( ) throws MIDletStateChangedException {
        }

        protected void pauseApp( ) {
         }

        protected void destroyApp(boolean unconditional)
         throws MIDletStateChangedException {
        }
}
```

Listing 3.1 MIDlet Code

The Application Management Software (AMS) is the environment in which a MIDlet is installed, started, stopped and uninstalled. The AMS is also sometimes called the Java Application Manager (JAM). The AMS creates each new MIDlet instance, and controls its state by directing a MIDlet to start, pause or destroy itself.
MIDlets are at any given time in one of three states: *Paused*, *Active*, or *Destroyed*. The state diagram in Figure 3.6 shows how these states are related and the legal state transitions.

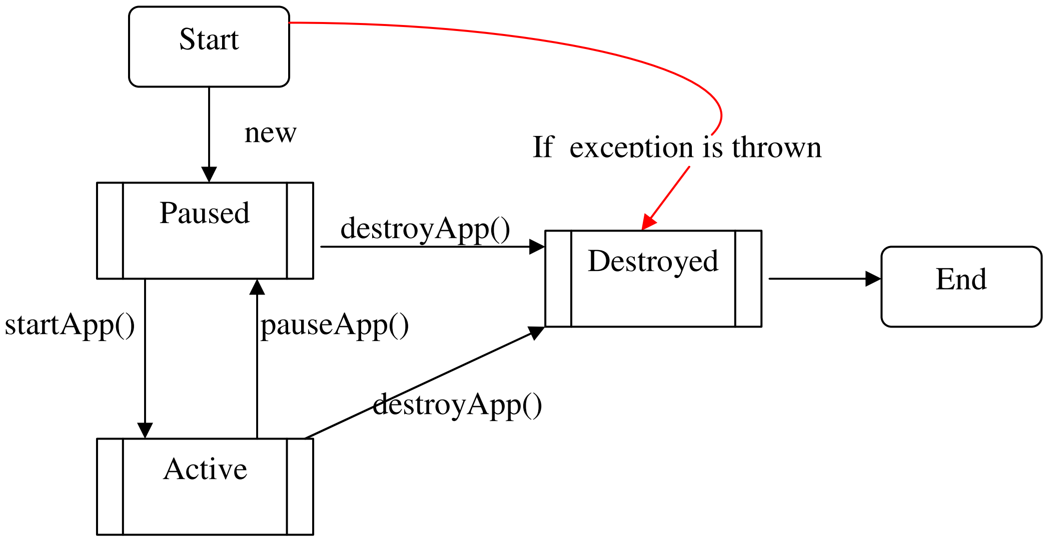


Figure 3.6 The lifecycle of a MIDlet.

When a MIDlet is instanciated it is in the *Pause* state. If an Exception occurs the status is changed to *Destroyed*. The MIDlet changes it status to *Active* through the call of the method *startApp()*. By calling the method *destroyApp* (boolean unconditional), the MIDlet is stopped by enforcement. The parameter set to false will lead to a *MIDletStateChangeException*. If the MIDlet handles this excpetion it will not change its status to *Destroyed* by enforcement.

The developers mainly concentrate on what the MIDlet has to do in the *Active* state and to write the code needed to let the MIDlet change into the *Paused* or *Destroyed* state. It is possible to collect a group of related MIDlets into a MIDlet suite. All of the MIDlets in a suite are packaged and installed onto a device as a single entity, and they can be uninstalled and removed only as a group. The MIDlets in a suite share, both static and runtime resources of their host environment. The Midlet application is ready to be deployed on a mobile phone when it went through different steps of development, which are shown in Figure 3.7.

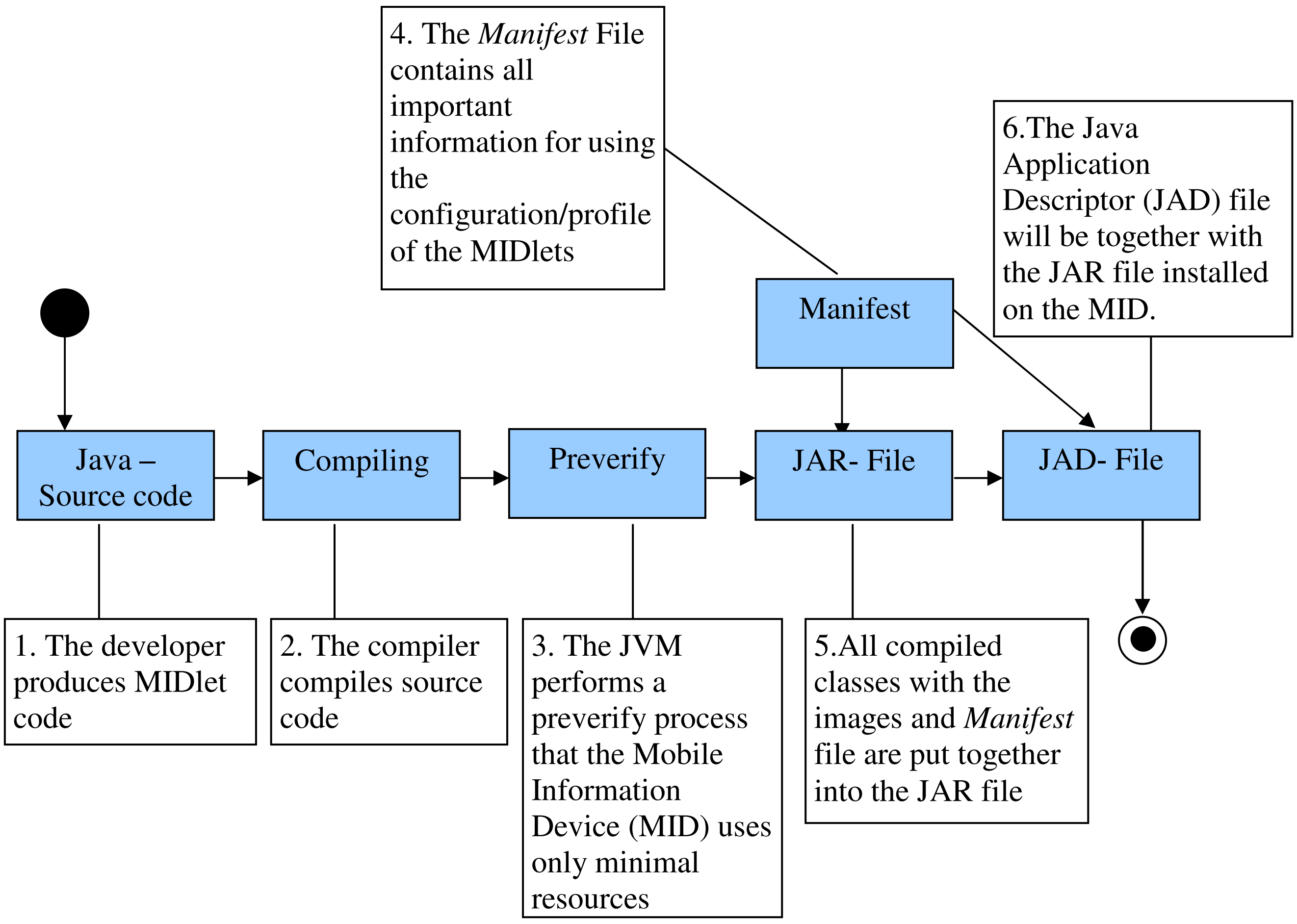

4. The Manifest File contains all important information for using the configuration/profile of the MIDlets
6.The Java Application Descriptor (JAD) file will be together with the JAR file installed on the MID.
Manifest
Java – Source code
Compiling
Preverify
JAR- File
JAD- File
1. The developer produces MIDlet code
2. The compiler compiles source code
3. The JVM performs a preverify process that the Mobile Information Device (MID) uses only minimal resources
5.All compiled classes with the images and Manifest file are put together into the JAR file

Figure 3.7 The MIDlet development process

First, the developer has to write the MIDlet code and then compile it using the J2SE which uses the J2ME/MIDP package as its bootclasspath and generates the *.class* files, also called bytecode of the application.

javac -bootclasspath *\lib\midpapi.zip* ***–d*** *.\tmpclasses* ***–classpath*** *.\tmpclasses .\src*.java*

Listing 3.2 Compile command

The source files of Listing 3.1 are compiled in the *src* directory and then saved into the *tmpclasses* directory. Listing 3.1 illustrates this compilation procedure.

Obfuscation, which is an operation to compress the files, is used after that to hold the size of files as small as possible. Listing 3.3 and Listing 3.4 show the Code before and after obfuscation.

```
public class Test {

        private String testname;

        public String getTestname() {
                return testname;
        }

        public void setTestname ( String newname ) {
                testname = newname;
        }
}
```

Listing 3.3 Code before obfuscation

The Obfuscator would change the code as follows:

```
public class a
{
        private String a;

        public String a() { return a; }

        public void b( String b ) { a = b; }
}
```

Listing 3.4 Code after obfuscation

The next step is to verify the bytecode. The bytecode verification process guarantees that an application cannot access memory spaces or use resources outside of its domain. Bytecode verification also prevents an application from overloading the Java language core libraries, a method that could be used to bypass other application-level security measures.
Due to the high computational overhead of this operation, however, MIDP JVMs do not perform complete bytecode verification at runtime. Instead, the application developer must preverify the classes on a development platform or staging area before deploying the application into mobile devices. The pre-verification process optimizes the execution flows, creates stackmaps containing catalogs of instructions in the application, and then adds the stackmaps to the pre-verified class files. At runtime, the MIDP JVM does a quick linear scan of the bytecode, matching each valid instruction with a proper stackmap entry. Listing 3.5 illustrates this step.

```
Preverify  -classpath \lib\midpapi.zip; .\tmpclasses -d .\classes .\tmpclasses
```

Listing 3.5 Preverify command

The preferified classes will be copied to the *classes* directory. The preferified classes are collected together with the *Manifest* file, which holds important information about the configuration version, profile version of the MIDlet, MIDlets name, version and vendor into a JAR file. Listing 3.6 shows an example of a *Manifest* file

```
MIDlet-1: test, , test.sipua.Test
MIDlet-Name: SIPUA
MIDlet-Vendor: Unknown
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0
```

Listing 3.6 Example of a *Manifest* file

The command to construct the JAR file is illustrated in Listing 3.7.

```
jar cmf Manifest.MF sipua.jar -c .\classes
```

Listing 3.7 Command to construct JAR files

At the end, there is a JAD file which holds information from the *Manifest* file and the size of JAR file,. An example pf a JAD file is shown in Listing 3.8.

```
MIDlet-1: test, , test.sipua.Test
MIDlet-Jar-Size: 100
MIDlet-Jar-URL: SIPUA.jar
MIDlet-Name: SIPUA
MIDlet-Vendor: Unknown
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0
```

Listing 3.8 Example of a JAD file

The JAR file together with the JAD file are deployed on the Mobile Information Device (MID). They are called Midlet Suite. Figure 3.8 shows two ways for deploying a MIDlet on a mobile phone.

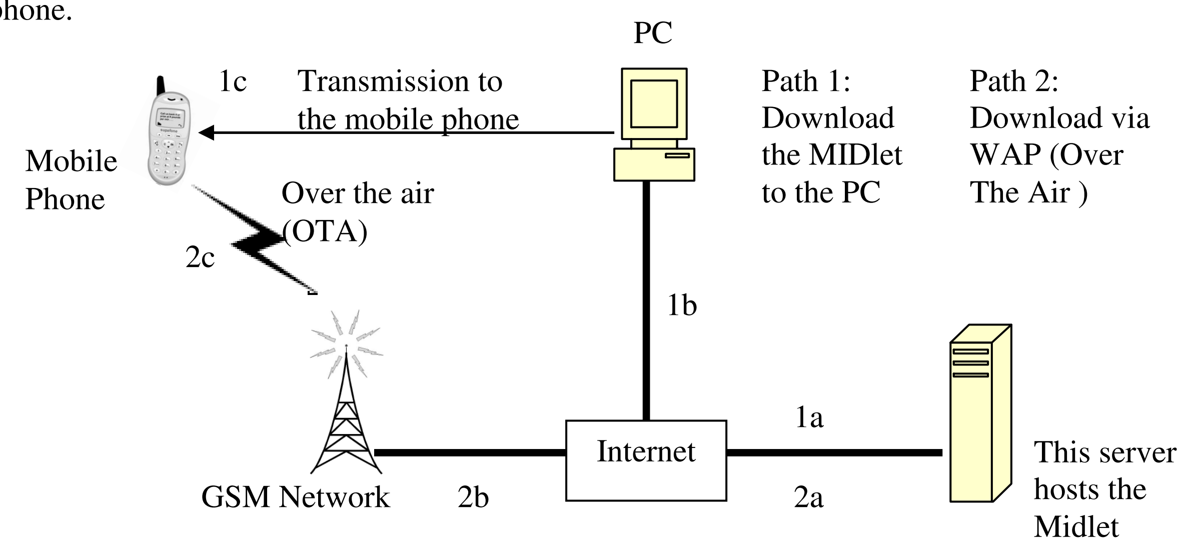

PC
Mobile Phone
1c
Transmission to the mobile phone
Path 1: Download the MIDlet to the PC
Path 2: Download via WAP (Over The Air)
Over the air (OTA)
2c
1b
1a
Internet
GSM Network
2b
2a
This server hosts the Midlet

Figure 3.8 Deploying a MIDlet on a mobile phone

There are two ways to install a MIDlet on a mobile phone. The first way is by a direct connection from a PC to the end device (path 1) and the second through a Wireless Markup Language (WML) site from Over The Air (OTA) or.

3.4.3 MIDP User Interface APIs

The MIDP User Interface (UI) consists of two types of APIs. One is the high-level and the other the low-level API. The low-level API is based on use of the abstract class Canvas. The high-level API's classes as *Alert*, *Form*, *List* and *TextBox* are extensions of the abstract class *Screen*. The high-level API classes are designed to provide abstractions and components that are highly portable, as the actual implementation takes care of aspects such as drawing, font characteristics, navigation and scrolling. The particular device's implementation of these classes performs the adaptation to its hardware and native UI look-and-feel.

The low-level API's *Canvas* class allows applications to have more direct control of the UI. It allows greater control of what is drawn on the display, and receiving low-level keyboard events. It is the application programmer's responsibility to ensure portability across MIDs with different characteristics (e.g. display size, colour vs. black-and-white and different keyboard types).

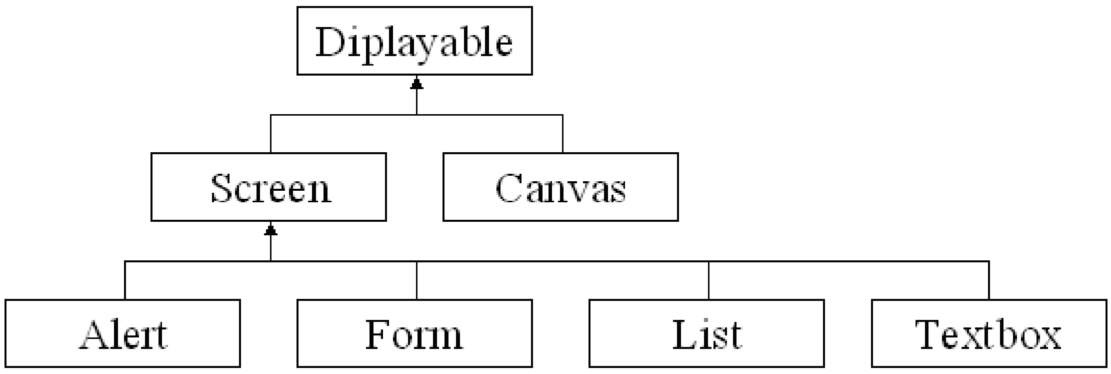


Figure 3.9 MIDP User Interface class hierarchy

3.4.4 Networking

One of the important aspects in J2ME is the network connectivity. Many advantages to the mobile phone adds the network capability. They allow to get use of the powerful network resources and to be able to make Internet connections. Most important is the capability to access networks wirelessly.

J2ME networking was designed to address the different needs of a wide spectrum of mobile devices. At the same time, the networking system must be device specific. To meet these needs, the concept of a Generic Connection Framework (GCF) was introduced as mentioned before. The idea of the GCF is to define the abstractions that cover the general aspects of networking and file Input/Output (I/O) in the form of Java interfaces.

Section 3.1 illustrates the available connection classes in the GCF. In this section I want to give an overview of the meaning about the different classes as shown in Table 3.2. This includes connections can be set up.

GCF Interface	**Purpose**
Connection	The most basic class for connections in the GCF. All other connection types extend *Connection*.
ContentConnection	Manages a connection, such as Hypertest Transfer Protocol (HTTP), for passing content, such as Hypertext Markup Language (HTML) or Extenible Markup Language (XML). provides basic methods for inspecting the content length, encoding and type of content.
Datagram	Acts as a container for the data passed on a *DatagraConnection*.
DatagramConnection	Manages a *DatagramConnection*.
InputConnection	Manages an input stream-based *Connection*.
OutputConnection	Manages an output stream-based *Connection*.
StreamConnection	Manages the capabilities of a stream. Combines the methods of both *InputConnection* and *OutputConnection*.
StreamConnectionNotifier	Listens to a specific port and creates a *StreamConnection* as soon as activity on the port is detected.

Table 3.3 Generic Connection Framework classes and their purpose

The *Connector* class is used to create instances of a connection protocol using one of *Connector*'s static methods. The *Connector* defines three variations of an *open()* method that return a *Connection* instance. One method takes as an parameter a *String* which is a Unified Resource Identifier (URI) and is composed of three parts: a scheme, an address, and a parameter list. The general form of the name parameter is as illustrated in Listing 3.9.

<scheme>:<address>;<parameters>

Listing 3.9 General form of a name parameter

The scheme identifies the protocol of the connection, for example http, file, datagram or others. The address part identifies to connect to : www.domain.org or file.txt and the resource parameters identify other information that is required by the protocol to establish a connection such as the connection speed. The parameters are specified as *name=value* pairs , when needed. Some examples of the name URI are shown .

- *http://www.google.com:8080*
- *socket://localhost:8080*
- *file:c:/filename.txt*
- *datagram://127.0.0.1:8099*
- *comm:0;baudrate=9600*

When the URI is passed to the *Connector.open()* method, the *Connector* parses the URI into its various parts , *<scheme>:<address>;<parameters>.*
The scheme, in combination with other information such as the root package name and a platform identifier, allows the *Connector* to determine the right *Connection* implementation to create.

Low level IP networking, which includes sockets (defined by *javax.microedition.io.StreamConnectionNotifier*) for client server communication, datagram (defined by *javax.microedition.io.DatagramConnection*), serial port, and file I/O communication. Socket-based communication belongs to the connection-oriented TCP/IP protocol. Datagram-based communication belongs to the connectionless UDP/IP protocol. UDP provides a way for applications to send encapsulated raw IP datagrams without having to develop a connection. Unlike the connection-oriented protocol, which requires source and destination addresses, the datagram only requires a destination address. It is possible to handle file I/O and allowing a MIDlet to register for network access to a local serial port.

Secure networking in J2ME involves additional interfaces available for secure communication. Secure interfaces are supported by HTTPS and Secure Socket Layer /Transport Layer Security (SSL/TLS) over the IP network.

The communication between a mobile device and a web server is based on the HTTP (Hypertext Transfer Protocol) and defined by the *javax.microedition.io.HttpConnection* class. HTTP is a connection-oriented request-response protocol in which the parameters of the request must be set before the request is sent.

Here are some examples of making different connections:

For HTTP-based communication:

Connection conn = Connector.open("http://www.yahoo.com");

For stream-based socket communication:

Connection conn = Connector.open("socket://localhost:9000");

For datagram-based socket communication:

Connection conn = Connector.open("datagram://:9000");

For serial port communication

Connection conn = Connector.open("comm:0;baudrate=9000");

For invoking file I/O communication

Connection conn = Connector.open("file://myfile.dat");

The *Connector.open()* method also accepts the access mode (values *READ*, *WRITE*, and *READ_WRITE*), and a flag to indicate that the caller wants a timeout notification.

- *open(String name)*
- *open(String name, int mode)*
- *open(String name, int mode, boolean timeouts)*

In secured networking, an *HttpsConnection* is returned from *Connector.open()* when an *https://* connection string is accessed. A *SecureConnection* is returned from *Connector.open()* when an *ssl://* connection string is accessed.
No matter what type of URL is used, invoking *Connector.open()* opens an input stream of bytes from the *Connection* to the *java.io.InputStream*. This method is used to read every character until the end of the file (marked by a *-1*). If an exception is thrown, the JVM closes the connection and streams **[IBMNETW]**.

3.5 The Wireless Toolkit

There are many different tools for developing mobile applications. One of them is the Wireless Toolkit from SUN Microsystems. The Wireless Toolkit (WTK) for J2ME developers offers a way to construct MIDlets automatically. Moreover, it includes an emulator where it is possible to test MIDlets.

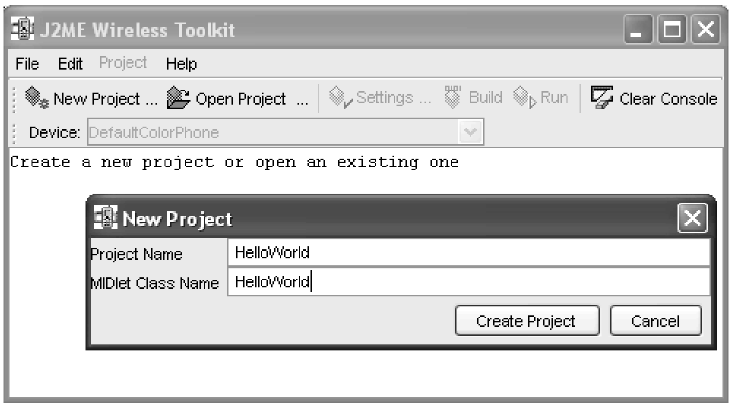
To illustrate this, an example of a HelloWorld MIDlet is shown in Listing 3.10:

```
// Import of the needed packages
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class HelloWorld extends MIDlet implements CommandListener {

        // Screen und Command-Objekt
        private Form myScreen;
        private Command EXIT_CMD = new Command( "Exit", Command.EXIT, 1 );

        // Methods invoked at start from the MIDlets
        protected void startApp() {
                myScreen = new Form( "MIDlet-Titel" );
                myScreen.addCommand( EXIT_CMD );
                myScreen.setCommandListener( this );
                myScreen.append( "Hello World!" );
                Display.getDisplay( this ).setCurrent( myScreen );
        }

        // Methods invoked when status changed to pause
        protected void pauseApp() {}

        // Methods invoked at end of the application
        protected void destroyApp(boolean unconditional) {}

        // Methods for the CommandListener-Interface
        public void commandAction( Command c, Displayable d ) {
                if ( c == EXIT_CMD ) {
                destroyApp( true );
                notifyDestroyed();
                }
        }
}
```

Listing 3.10 HelloWorld-MIDlet

The abstract methods *startApp()*, *pauseApp()* and *destroyApp()* are inherited from the MIDlet class. Therefore they must be implemented in the *HelloWorld* class. At start of the MIDlet the method *startApp()* is invoked. This method constructs the *Form* and adds a text to it. After that, the static method *Display.getDisplay()* calls the Display-Object of the MIDlets. The screen object is set to the current screen and painted by calling the method *setCurrent().* The method *commandAction()* is a method of the *CommandListener* Interface that needs to be implemented in the *HelloWorld* class. This method is invoked when for example a button is pressed.

At this moment, the current screen and the pressed button are imput parameters for the method. The *Command* object *EXIT_CMD* is the button needed to terminate a Midlet. At start of the *Ktoolbar,* the subsequent window is shown. In this window, a „New Project“ can be chosen to create a new project. We will create the project „HelloWorld“ as illustrated in Figure 3.10.



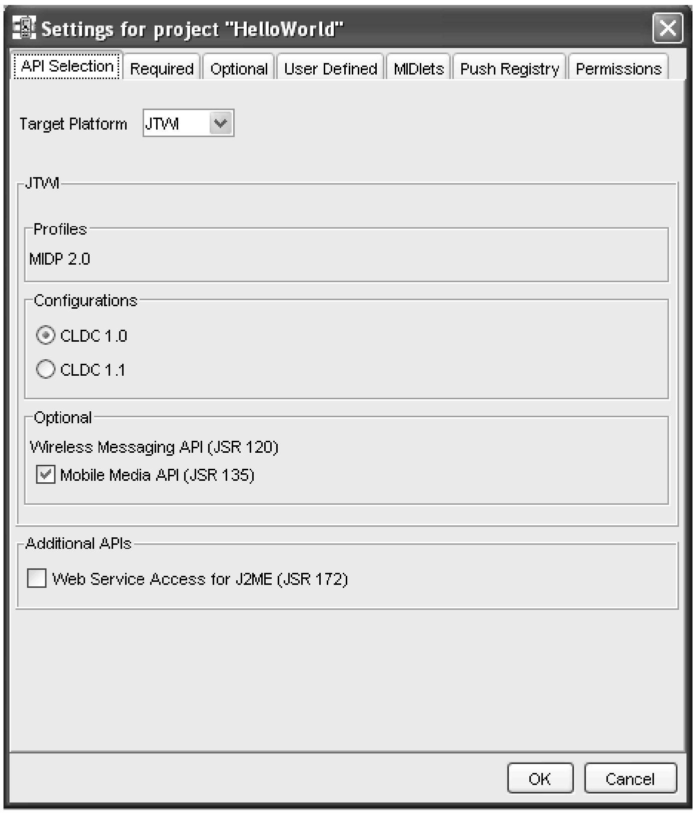
Device: DefaultColorPhone

Create a new project or open an existing one

Project Name: HelloWorld
MIDlet Class Name: HelloWorld

Figure 3.10 Ktoolbar from the Wireless Toolkit of SUN Microsystems

After pressing on the „Create Project“ button, a window pops up. It allows tp provide information about the MIDlet like MIDP and CLDC packages versions. This depends, on the end device, on which the application should run. The UI for this configuration process is shown in Figure 3.11.


Settings for project "HelloWorld"

API Selection | Required | Optional | User Defined | MIDlets | Push Registry | Permissions

Target Platform: JTWI

JTWI

Profiles: MIDP 2.0

Configurations: [✓ CLDC 1.0] CLDC 1.1

Optional: Wireless Messaging API (JSR 120), [✓ Mobile Media API (JSR 135)]

Additional APIs: ☐ Web Service Access for J2ME (JSR 172)

Figure 3.11 Project Configuration

Those data are saved in the *Manifest.MF* file and in the *HelloWorld.jad* in the *bin* directory.

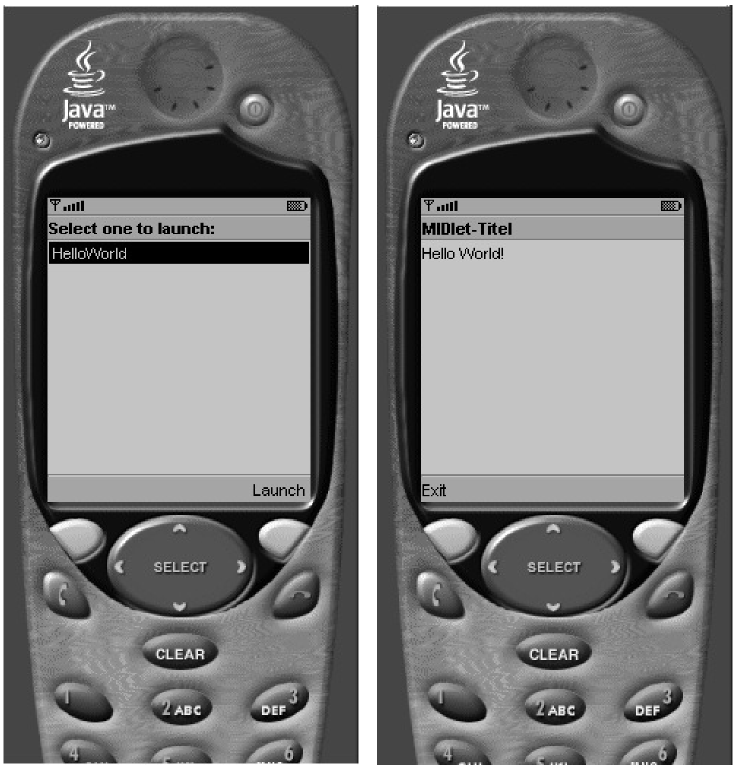
By creating the project, a new directory with the project name is created under *WTKHOME\apps*\. In this project directory there are four other directories which are: *bin*, *lib*, *res* und *src*. Table 3.4 gives an overview about the WTK directories.

Directory	**Description**
bin	KToolbar saves the *Manifest.MF*, *JAD* and *JAR* files in this directories
lib	This directory contains external libaries for the application
res	All pictures and icons should be placed here so that Ktoolbar can include it to the Java Archive (JAR)
src	This directory holds the source code
classes	Here the preverified classes are placed by the KToolbar
tmpclasses	KToolbar saves the compiled classes, which are still not preverified, in this directory
tmplib	This directory is used for library compilations

Table 3.4: Directories in each WTK project

Now we can build our source code. The compiled code will be automatically preverified and packed into a *Hello-World.jar* archiv. After this step finished, it is possible test and execute the MIDlet.

When pressing „Run“ in the Ktoolbar the left hand side of Figure 3.12 is shown. Launching the MIDlet, it shows the expected “HelloWorld!” text.



Select one to launch:
HelloWorld
Launch

MIDlet-Titel
Hello World!
Exit

Figure 3.12 MIDlet example running in the emulator of the KToolbar

There are nowadays different higher-larger APIs from hardware vendors which are very useful. On the website of SUN, those APIs could be found and used for mobile application development. The following enumeration lists the most important representatives:

- **Mobile Media API** for playing video, sound, and other media, I used this package in this master thesis for voice capturing and play back.
- **Wireless Messaging API** which provides platform independent access to wireless communication resources like Short Message Service (SMS).
- **Bluetooth API** for communication with other bluetooth devices
- **Java Speech API** for incorporate speech technology into user interfaces for applications. It supports command and control recognizers, dictation systems and speech synthesizers.
- **Mobile Game API** for game development**.**
- **Mobile 3D Graphics API** for displaying 3D objects
- **Location API** for location based services

3.6 Critical Factor for Application Development

During development of Java applications for J2ME, different kind of problems arise because of the different end devices. Those problems are the missing support of double operations, few memory, slow transfere rates, slow CPU, and the missing possibility to serialise objects. A developer must be in complete awareness of that and applications running in an emulator could also fail to run in a real mobile phone at the end.
A real success is after testing it on different kind of mobile phones or just designing an application for certain series of mobile phones and testing it for that group on a real mobile phone which could be very expensive.

4 Session Initiation Protocol

This section describes the Session Initiation Protocol (SIP) **[RFC3261]**, its functionality and importance. SIP is so important mainly because of its mobility support. Types of mobility are device or terminal mobility, user mobility and finally service mobility. Terminal mobility allows the device to change its IP subnet and to be still reachable (e.g a redirect). Session mobility allows a user to have the same session although changing the end device. Personal mobility allows the user to be reachable on the same logical address on many devices. Finally, the Service mobility allows the user to enter to his services by changing its network or his end device.
This section goes mainly through different SIP scenarios, the SIP Messages, SIP Header fields, explains the SIP User Agent components, the Session Description Protocol (SDP) and finally some Java API for SIP for mobile devices.

4.1 What is SIP?

The SIP protocol is initially standardized by the Internet Engineering Task Force (IETF) Multiparty Multimedia Session Control (MMUSIC) Working Group. It is an application-layer signaling control protocol that sets up real-time multimedia sessions between groups of participants and manages the creation, modification and termination of those sessions over packet-based networks. A set of compatible media types used in the sessions are included by the sessions in the session description part.
A main advantage of SIP is the mobility of the participants by proxying and redirecting requests to the user's current location. The exact current location is known by the registration of each user’s current location to a registrar server.
The sessions could include multimedia conferences, distance learning, Internet telephony and similar applications.
To provide complete service to a mobile user, SIP should be used in conjunction with other protocols. The Real-time Transport Protocol (RTP) and the Session Description Protol (SDP) are used in the implementation of the SIP User Agent for a mobile device although other protocols could have been used. An example which is usually used in conjuction is the Real-time Transport Control Protocol (RTCP). In this work, it has not been used, due to the limited capability of the mobile devices.
In this thesis, I will focus on SIP's capabilities for real time audio calls between two mobiles, and how it sets up calls that then use RTP to actually send the voice data between mobiles. This application could also send Instant Messages. Therefore, SIP provides two modes. One of them is the Pager Mode, like sending SMS, and the other one is the Session Mode, that behaves like a chat application.

Figure 4.1 shows the network layers and were the SIP signaling is located in those layers.

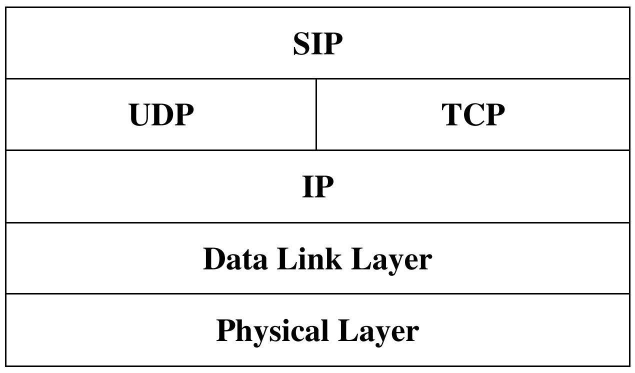


Figure 4.1 Session Initiation Protocol Stack

In IP, there are two different phases of a voice call. The first phase is called the "call setup," and includes all of the details needed to connect two telephones. Once the call has been setup, the phones enter a "data transfer" phase of the call using different protocols to actually move the voice packets between the two phones.
SIP is a very flexible protocol and is designed to be independent of the lower-layer transport protocol. For example, in addition to simple telephone calls, SIP can also be used to set up video and audio multicast meetings, or instant messaging conferences. SIP does more than just handle call setup. The subsequent Table 4.1 shows the five major functions of SIP.

Function	**Description**
User location	determination of the end system to be used for communication
User capabilities	determination of the media and media parameters to be used.
User availability	determination of the willingness of the called party to engage in communications.
Call setup	"ringing", establishment of the call at both, called and calling party.
Call handling	including transfer and termination of calls.

Table 4.1: Functionality of SIP

SIP can run over IPv4 **[RFC2794]** and IPv6 **[RFC1924]** and it can use either TCP or UDP. The most common implementations use IPv4 and UDP. This minimizes the overhead and therefore it performs best.
SIP devices can talk directly to each other but generally SIP traffic passes intermediary network components like SIP proxies. These SIP proxies only handles the signalling messages but once the call is set up, the multimedia data (voice data) packets are sent directly between the SIP terminals without passing a proxy.
Recent mobile devices used to have their own telephone number. In SIP, an end station (caller or callee) is identified by a unique SIP Uniform Resource Identifier (URI) . These URIs are used in the SIP protocol. They have a similar form to an email address, typically containing a username and a host name.

4.2 Call Flow Examples

Now I explain different call flow examples. I start with a basic example without and with SIP proxy, then I will give a SIP registration example and finally I will describe SIP operation using a redirect server.

4.2.1 Basic Session Establishment

The subsequent simple example explains the basic SIP operation. A SIP message exchange between two users, Samya and Klaus shows Figure 4.2.

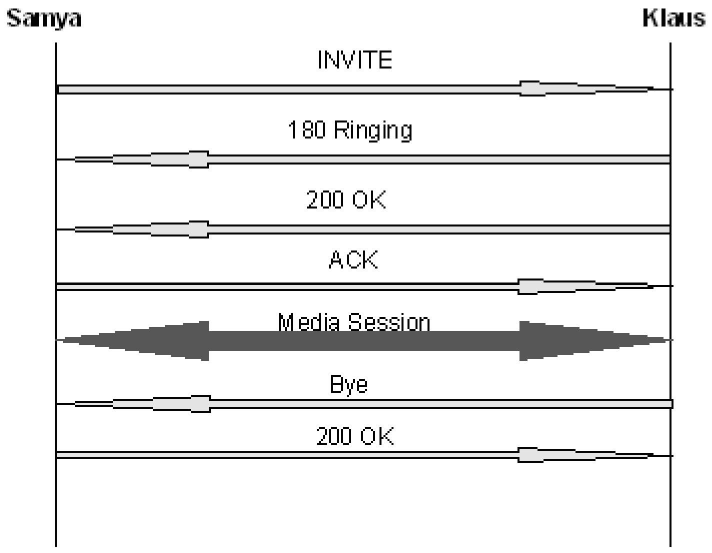


Figure 4.2: Basic SIP session setup

Samya uses a SIP application to call Klaus on his SIP aware mobile phone. Samya calls Klaus using his SIP URI identity. It is sip: *Klaus@ikn.tuwien.ac.at*, where *ikn.tuwien.ac.at* is the domain of Klaus's SIP service provider. Samya has a SIP URI of *sip: Samya@hotmail.com*. Samya starts by sending an *INVITE* request with Klaus's SIP URI. *INVITE* is an example of a SIP method. SIP is mainly based on requests and responses. Klaus's SIP mobile receives the *INVITE* and alerts Klaus about the incoming call from Samya so that Klaus can decide whether to answer the call. Klaus's SIP mobile indicates this in a *180 (Ringing)* response. When Samya's mobile receives the *180 (Ringing)* response, it passes this information to Samya, perhaps using an audio ringback tone or by displaying a message on Samya's screen.

As in the figure 4.2, Klaus decides to answer the call. By doing this, his SIP mobile sends a *200 (OK)* response to indicate that the call has bee. The *200 (OK)* contains a message body with the SDP media description of the type of session that Klaus is willing to establish with Samya. As a result, there is a two-phase exchange of SDP messages: Samya sent one to Klaus, and Klaus sent one back to Samya. This two-phase exchange provides basic negotiation capabilities and is based on a simple offer/answer model of SDP exchange. If Klaus did not wish to answer the call or was busy on another call, an error response would have been sent instead of the *200 (OK),* which would have resulted in no media session being established.

Finally, Samya's SIP software on the mobile sends an acknowledgement message, *ACK*, to Klaus's SIP mobile to confirm the reception of the final *200 (OK)* response. In this example, the *ACK* is sent directly from Samya's mobile to Klaus's SIP mobile.
Samya and Klaus's media session has been estabilished successfully. Therefore, they send media packets using the format to which they agreed in the exchange of SDP. In general, the end-to-end media packets take a different path than the SIP signaling messages. At the end of the call, Klaus disconnects first and generates a *BYE* message. This *BYE* is routed directly to Samya's mobile. Samya confirms the receipt of the *BYE* with a *200 (OK)* response, which terminates the session and the *BYE* transaction. No *ACK* is sent - an *ACK* is only sent in response to a response to an *INVITE* request.

4.2.2 SIP Registration

This registration example learns about the location of the user with address *klaus@ikn.tuwien.ac.at* and binds this address to user’s location to which he wants the requests to go which is 195.37.78.173.

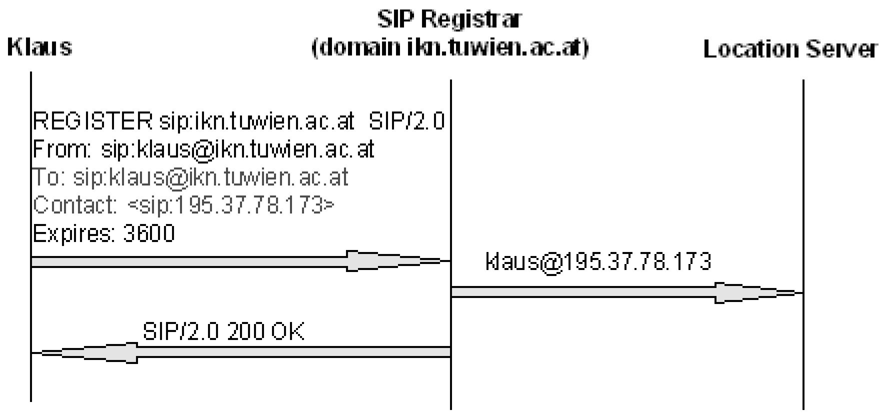

Klaus
SIP Registrar
(domain ikn.tuwien.ac.at)
Location Server
REGISTER sip:ikn.tuwien.ac.at SIP/2.0
From: sip:klaus@ikn.tuwien.ac.at
To: sip:klaus@ikn.tuwien.ac.at
Contact: <sip:195.37.78.173>
Expires: 3600
klaus@195.37.78.173
SIP/2.0 200 OK

Figure 4.3: Registration scenario

As shown in Figure 4.3, for registration the following steps have to be done:

1) First the User Agent has to send the required data which are *From*, *To*, *Contact*, *Expires* fields with the “*REGISTER*” SIP method, to the SIP registrar server as shown in Listing 4.1.

```
REGISTER sip:ikn.tuwien.ac.at  SIP/2.0
From: sip:klaus@ikn.tuwien.ac.at
To: sip:klaus@ikn.tuwien.ac
Contact: <sip:195.37.78.173>
Expires: 3600
```

Listing 4.1 SIP REGISTER request

2) This server sends the important information to the Location Server to save that *klaus@ikn.tuwien.ac.at* will be found at address 195.37.78.173.

3) After successfully saving this data the registrar server sends a SIP *200 OK* response back to the User Agent of Klaus. Listing 4.2 illustrates a successful response.

```
SIP/2.0 200 OK
To:Klaus <sip:kaus@ikn.tuwien.ac.at>;tag=a6c85cf
From:Registrar <sip: registrar@ikn.tuwien.ac.at>;tag=1928301774
Expires: 3600
```

Listing 4.2 SIP succesful response

Now any request to *klaus@ikn.tuwien.ac.at* goes to a proxy server, which binds this name to its current location and the request will be forwarded to the current right location of Klaus. Klaus has always to send a register request in case his current location changes or expiration of registration has reached. Otherwise, future requests will not reach him.

4.2.3 Simple Session Establishment with SIP Proxy Server

The subsequent simple example explains the basic SIP operation with proxies. A SIP message exchange between two users, Samya and Klaus shows Figure 4.2 The proxies are mainly responsible to forward the reuqsts to the appropriate terminal.

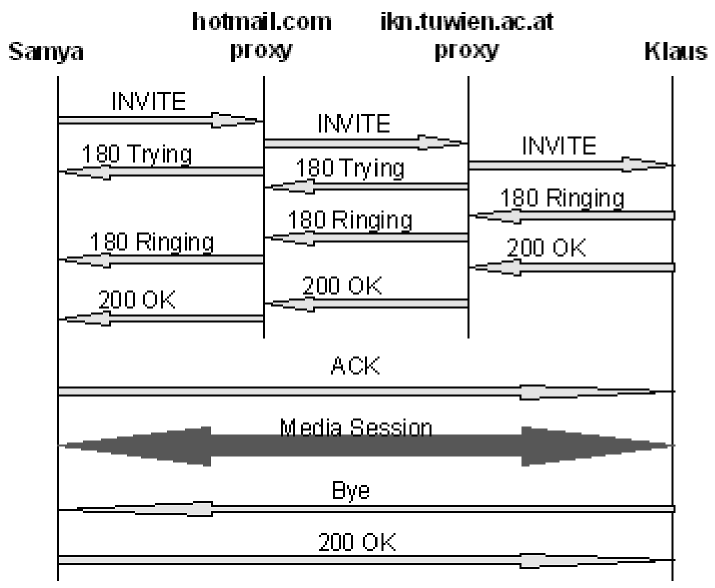


Figure 4.4: SIP session setup example with SIP proxy server

Samya uses a SIP application to call Klaus on his SIP aware mobile phone. There exist two SIP proxy servers that facilitate the session establishment. Samya calls Klaus using his SIP URI. It is *sip:Klaus@ikn.tuwien.ac.at*, where *ikn.tuwien.ac.at* is the domain of Klaus's SIP service provider. Samya has a SIP URI of sip: *Samya@hotmail.com.*

Samya starts by sending an *INVITE* request with Klaus's SIP URI to the proxy server of her domain. *INVITE* is an example of a SIP method. The *hotmail.com* SIP server is a type of SIP server known as a proxy server. A proxy server receives SIP requests and forwards them. In this example, the proxy server receives the *INVITE* request and sends a *100 (Trying)* response back to Samya. The *100 (Trying)* response indicates that the *INVITE* has been received and that the proxy is working to forward the *INVITE* to the destination. SIP is mainly based on requests and responses.
The hotmail.com proxy server tries to find the proxy server at *ikn.tuwien.ac.at*, possibly by Domain Name Service (DNS) lookup to find the SIP server that serves the *ikn.tuwien.ac.at* domain. The result is that it gets the IP address of the *ikn.tuwien.ac.at* proxy server and forwards, or proxies, the *INVITE* request to this IP address. The ikn.tuwien.ac.at proxy server checks the location server where, Klaus is and forwards the request to his UA.
Klaus's SIP mobile receives the *INVITE* and alerts Klaus to the incoming call from Samya so that Klaus can decide whether to answer the call. Klaus's SIP mobile indicates this in a *180 (Ringing)* response, which is routed back through the two proxies in the reverse direction. When Samya's mobile receives the *180 (Ringing)* response, it passes this information to Samya, perhaps using an audio ringback tone or by displaying a message on Samya's screen.
As shown from the figure, Klaus decides to answer the call. By doing this, his SIP mobile sends a *200 (OK)* response to indicate that the call has been answered. The *200 (OK)* contains a message body with the SDP media description of the type of session that Klaus is willing to establish with Samya. As a result, there is a two-phase exchange of SDP messages: Samya sent one to Klaus, and Klaus sent one back to Samya. This two-phase exchange provides basic negotiation capabilities and is based on a simple offer/answer model of SDP exchange. If Klaus did not wish to answer the call or was busy on another call, an error response would have

been sent instead of the 200 (OK), which would have resulted in no media session being established.

Finally, Samya's SIP software on the mobile sends an acknowledgement message, *ACK*, to Klaus's SIP mobile to confirm the reception of the final *200 (OK)* response. In this example, the *ACK* is sent directly from Samya's mobile to Klaus's SIP mobile, bypassing the two proxies. This occurs because the endpoints have learned each other's address from the *Contact* header fields through the *INVITE/200 (OK)* exchange, which was not known when the initial *INVITE* was sent.
Samya and Klaus's media session has been estabilished successfully. Therefore, they send media packets using the format to which they agreed in the exchange of SDP. In general, the end-to-end media packets take a different path than the SIP signaling messages. At the end of the call, Klaus disconnects first and generates a *BYE* message. This *BYE* is routed directly to Samya's mobile, again bypassing the proxies. Samya confirms the receipt of the *BYE* with a *200 (OK)* response, which terminates the session and the *BYE* transaction.

A registration occurs when a client needs to inform a proxy or redirect server of its location. During this process, the client sends a *REGISTER* request to the registrar server and includes the address (or addresses) at which it can be reached. Registration is another common operation in SIP. In our example, it would be that the *ikn.tuwien.ac.at* server can learn the current location of Klaus.

Upon initialization, and at periodic intervals, Klaus's SIP mobile sends *REGISTER* messages to a server in the *ikn.tuwien.ac.at* domain known as a SIP registrar. The *REGISTER* messages associate Klaus's SIP URI (*sip:Klaus@ikn.tuwien.ac.at*) with the terminal where he currently resides in (conveyed as a SIP URI in the *Contact* header field).
The registrar writes this association, also called a binding, to a database, called the location service, where it can be used by the proxy in the *ikn.tuwien.ac.at* domain. Often, a registrar server for a domain is co-located with the proxy for that domain. It is an important concept that the distinction between types of SIP servers is logical, not physical.
Klaus is not limited to registering from a single device. For example, both, his SIP mobile at home and the one in the office, could send registrations. This information is stored together in the location service and allows a proxy to perform various types of searches to locate Klaus. Similarly, more than one user can be registered on a single device at the same time.

Figure 4.5 shows another example of SIP operation in proxy mode. The condition to have operation in proxy mode is that every User Agent has to tell the proxy (registrar) at which IP address to be reached.

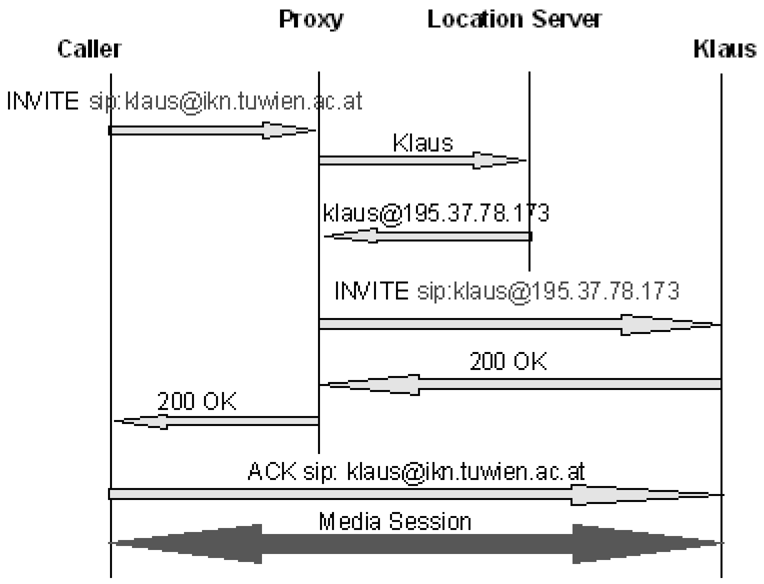

Proxy
Location Server
Caller
Klaus
INVITE sip:klaus@ikn.tuwien.ac.at
Klaus
klaus@195.37.78.173
INVITE sip:klaus@195.37.78.173
200 OK
200 OK
ACK sip: klaus@ikn.tuwien.ac.at
Media Session

Figure 4.5: Proxy mode scenario

The detail of the messages will be explained in a subsequent section. In this section, I will just focus on the establishment of a general call.

1. The User Agent sends a domain name server query request to get the IP address of the proxy server of *ikn.tuwien.ac.at*.

2. The User Agent then sends an *INVITE* request to the proxy server. Listing 4.3 illustrates the *INVITE* request.

   INVITE sip:klaus@ikn.tuwien.ac.at
   From: sip:Caller@sip.com
   To: sip:klaus@ ikn.tuwien.ac.at
   Call-ID: 345678@sip.com

   Listing 4.3 SIP INVITE request

   In this example caller, which can be found at *sip.com* wants to reach Klaus and does not know his current location, so he sends the request to the proxy, which is responsible for *ikn.tuwien.ac.at*.

3. The SIP proxy server asks the location server on which IP address Klaus could be reached.
4. The loacation server sends the IP address to reach Klaus to the SIP proxy server.

5. Now, the Proxy is able to forward the *INVITE* request. This is illustrated in Listing 4.4.

INVITE sip:klaus@195.37.78.173
From: sip:Caller@sip.com
To: sip:klaus@ikn.tuwien.ac.at
Call-ID: 345678@sip.com

Listing 4.4 SIP INVITE request

6. After Klaus got the request, he answers with a *200 OK* message as illustrated in Listing 4.5.

SIP/2.0 200 OK
From: sip:Caller@sip.com
To: sip:klaus@ikn.tuwien.ac.at
Call-ID: 345678@sip.com

Listing 4.5 SIP 200 OK response

7. The SIP proxy server forwards this *200 OK* message back to the caller.

SIP/2.0 200 OK
From: sip:Caller@sip.com
To: sip:klaus@ikn.tuwien.ac.at
Call-ID: 345678@sip.com

8. Now the Caller acknowledges that he got the *200 OK* message by sending a *ACK* request.

ACK sip:klaus@ikn.tuwien.ac.at

9. Starting from this moment, both end phones know the locations of each other and exchange of data can start directly between them, where by the Caller is reached on Caller@sip.com and Klaus is reached on klaus@195.37.78.173.

4.2.4 Call Establishment with SIP Redirect Server

In the subsequent Figure 4.6 a redirect example is shown.

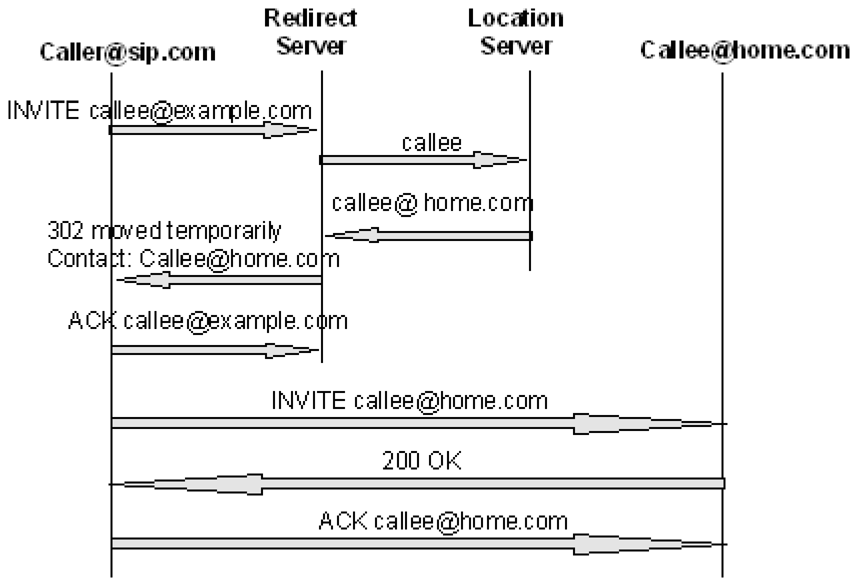


Figure 4.6: Redirect Mode Scenario

1) In this example, Caller, which can be found at *sip.com* wants to reach *Callee@home.com* and sends this request to the SIP redirect sewrver.The User Agent sends an INVITE request to a SIP Server, which acts as redirect server.

2) The redirect server asks the location server, where the Callee can be reached.

3) The loacation server sends the answer, which means the current address of the Callee to the redirect server.

4) Now the redirect server is able to send back the current address telling the Caller that Callee has moved and indicates the new domain name.

5) After the Callee gets the request, he sends a new *ACK* request to the redirect server.

6) Now, the Caller can reach Callee at his current address sending an *INVITE* request.

7) The Callee answers with a *200 OK*.

8) Finally, the Caller sends an ACK request at this moment, both terminals know the location of each other and an exchange of data can start directly between them. The Caller is reached on *Caller@sip.com* and the Callee is reached on *Callee@home.com.*

4.3 SIP Messages

There are two types of SIP messages, requests and responses. Clients originate requests and send them towards a server. Responses are the answers from a server to a client.

4.3.1 Requests

A User Agent can establish, modify or terminate a session using requests. Table 4.2 shows the main SIP request methods.

Message	**Description**
INVITE	Indicates a user or service is being invited to participate in a call session.
ACK	Confirms that the client has received a final response to an INVITE request
BYE	Terminates a call and can be sent by either the caller or the callee.
CANCEL	Cancels any pending searches but does not terminate a call that has already been accepted.
OPTIONS	Queries the capabilities of servers.
REGISTER	Registers the address listed in the To header field with a SIP server. Creates binding location.

Table 4.2: SIP request methods

4.3.2 Responses

Responses inform about the success of requests.There exist two different types of responses and six response classes.The first response type is the provisional response (1xx class). A provisional response shows the progress of the request. The second type is the final response to terminate SIP transactions. Table 4.3 enumerates the SIP response types and gives some examples.

Type	**Message**	Description
Provisional	1xx	Informational Responses • 100 Trying • 180 Ringing (processed locally) • 181 Call is Being Forwarded
Final	2xx	Successful Responses • 200 OK

	3xx	Redirection Responses • 300 Multiple Choices • 301 Moved Permanently • 302 Moved Temporarily
	4xx	Client Failure Responses • 400 Bad Request • 401 Unauthorized • 482 Loop Detected • 486 Busy Here
	5xx	Server Failure Responses • 500 Internal Server Error
	6xx	Global Failure Responses • 600 Busy Everywhere

Table 4.3: SIP response classes

4.4 SIP Header Fields

Every SIP message consists of three main parts, which are the start line, header and the body. The start line shows the type of a message. This could be either the request information or the status code of the response. In case of a request, a request URI is included to indicate to whom this user or service belongs. In case of a response, a status code and the realated text is shown. Moreover, the Protocol version is included at this start line.

The header part is used to show information about the message. A header could contain more than one line and some header fields could appear more than once.

The body contains information about session characteristics. This contains for example used codecs for audio or video could be shown. Message bodies may exist in requests as well as in responses. SDP or Multiple Internet Mail Extensions (MIME) are examples of body types.Listing 4.1 shows an example of INVITE message.

```
INVITE sip:klaus@ikn.tuwien.ac.at SIP/2.0
Via: SIP/2.0/UDP test.hotmail.com;branch=z9hG4bK776asdhds
Max-Forwards: 70
To: Klaus <sip:klaus@ikn.tuwien.ac.at >
From: Samya <sip:samya@hotmail.com>;tag=1928301774
Call-ID: a84b4c76e66710@test.hotmail.com
CSeq: 314159 INVITE
Contact: <sip:samya@test.hotmail.com>
Content-Type: application/sdp
Content-Length: 142
v=0
o=UserA 2890844526 2890844526 IN IP4 here.com
s=Session SDP
c=IN IP4 100.101.102.103
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Listing 4.1 SIP INVITE message

The first line of the message contains the method name as described, which is here *INVITE*. The lines that follow are a list of header fields. Table 4.4 shows a minimum of the required set.

Header Field	**Description**
Via	contains the routing information important to route responses. It also contains a branch parameter that identifies this transaction.
To	contains a display name and a SIP or SIPS URI towards which the request was originally directed.
From	contains a display name and a SIP or SIPS URI that indicate the originator of the request.This header field also has a tag parameter containing a random string (1928301774 in Listing 4.1) that was added to the URI by the User Agent. It is used for identification purposes.
Call-ID	contains a globally unique identifier for this call, generated by the combination of a random string and the phone's host name or IP address.
CSeq or Command Sequence	contains an integer and a method name. The CSeq number is incremented for each new request within a dialog and is a traditional sequence number.
Contact	contains a SIP or SIPS URI that represents a direct route to the contact, usually the username at a fully qualified domain name (FQDN) or the IP addresses. The Contact header field tells others where to send future requests.
Max-Forwards	serves to limit the number of SIP aware devices a request can pass on the way to its destination. It is decremented by one at each hop.

Content-Type	contains a description of the message body.
Content-Length	contains the message body count type.

Table 4.4: SIP header fields

The body of a SIP *INVITE* message contains a description of the session, encoded in some other protocol format as the Session Description Protocol (SDP) **[SDP]**. In this protocol the type of media, codec and sampling rate are included. The body part will be explained later in detail in Section 4.8.

4.5 SIP Components

In this section I will explain the different types of SIP servers and their functionality.

4.5.1 User Agent

A User Agent consists of an end system and contains a User Agent Client (UAC) for generating requests, and a User Agent Server (UAS) for generating responses. A UAC issues a request based on some external actions as for example a user is clicking on a button and pocesses the respective response. A UAS is capable of receiving a request and generating an appropriate response.

A UAC sends requests, which may pass through some proxies to be forwarded to a UAS. When the UAS generates a response, it is sent to the UAC via the same proxies.
Whether the request or response is inside or outside of a dialog (which represents a peer-to-peer relationship between User Agents and are established by specific SIP methods, such as *INVITE* and the request methods are playing an important role in the procedures of the UAC and UAS.

4.5.2 User Agent Server

A User Agent Server (UAS) is a server which is generating responses to SIP requests (of UACs). UASs should process the requests in the following order:

- Starting with authentication
- Method inspection
- Header inspection
- Content processing
- Processing the request
- Generating the response

4.5.3 SIP Proxy Server

A SIP proxy server operates as a server as well as a client to make requests for other clients or servers. The requests are either processed intern or forwarded. A proxy server interprets, rewrites or translates the message before forwarding it.

4.5.4 SIP Redirect Server

A SIP redirect server accepts SIP requests, maps the address into new addresses, and returns these addresses to the client. In other words, it is a User Agent Server that generates responses to requests it receives, directing the client to contact an alternate set of URIs. They do not initiate requests and do not accept calls. They require lower state overhead than proxy servers due to fewer messages to process. The redirect server offers services which are client device dependent.
In some architectures, it may be desirable to reduce the processing load on proxy servers that are responsible for routing requests, and improve signaling path robustness, by relying on redirection. Redirection allows servers to push routing information for a request back in a response to the client, thereby taking themselves out of the loop of further messaging for this transaction while still aiding in locating the target of the request. When the originator of the request receives the redirection, it will send a new request based on the URI(s) it has received. By propagating URIs from the core of the network to its edges, redirection allows for considerable network scalability.

4.5.5 SIP Registrar Server

A SIP registrar server is a server that accepts *REGISTER* requests and places the information it receives in those requests to the location service for the domain it handles.
SIP offers a discovery capability. If a user wants to initiate a session with another user, SIP must discover the current host(s) at which the destination user is reachable. This discovery process is frequently accomplished by SIP network elements such as proxy servers and redirect servers which are responsible for receiving a request, determining where to send it based on knowledge of the location of the user, and then sending it there. To do this, SIP network elements consult an abstract service known as a location service, which provides address bindings for a particular domain. These address bindings map an incoming SIP URI, *sip:klaus@tuwien.ac.at*, for example, to one or more URIs that are somehow closer to the desired user, *sip:klaus@ikn.tuwien.ac.at*, for example. Ultimately, a proxy will consult a location service that maps a received URI to the user agents at which the desired recipient is currently residing.

4.6 Session Description Protocol

SDP is the protocol **[SDP]** used to describe multimedia sessions, and multimedia session invitations. A multimedia session is defined, for these purposes, as a set of media streams that exist for a duration of time.
SDP usually includes session information as session name, purpose and the time the session is active and exists in the body of a SIP *INVITE* or *200 (OK)* message.
Since the resources necessary for participating in a session may be limited, it would be useful to include the following additional information:

- Information about the bandwidth to be used by the session.
- Contact information for the person responsible for the session.

To the media information belongs the following:

- Type of media, such as video and audio.

- Transport protocol, such as RTP/UDP/IP
- Media format, such as H.261 video or MPEG video.

The subsequent example shows a SDP payload which means: “receive RTP packets encoded audio in PCMU 8000Hz on IP Address 100.101.102.103:49172”

```
v=0
o=UserA 2890844526 2890844526 IN IP4 here.com
s=Session SDP
c=IN IP4 100.101.102.103
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

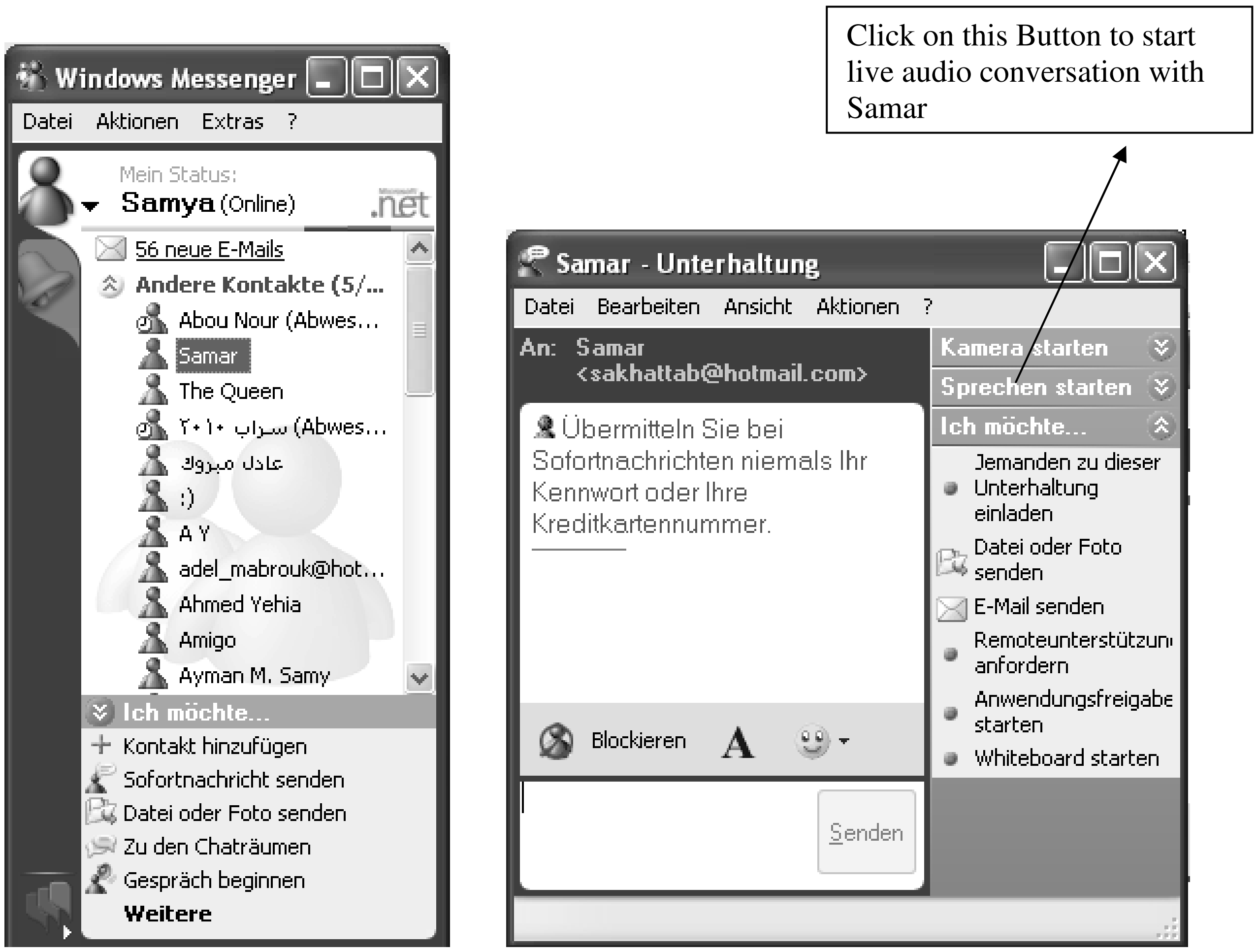
Listing 4.2 Session Description Protocol body

4.7 Examples for SIP User Agents

Sip User Agents are used in many chat applications, but the end user does not know that SIP is used as the underlying protocol. Here in this section, I explain an example of an application which is using SIP as its underlying protocol.

4.7.1 The Microsoft Messenger

Instant messenger programsnare widley spread on the Internet. Their main job is the capability of sending short text messages. Many applications are expanded now to support also live audio and video conversations. There are many vendors which offer this service. One of the famous one is the Microsoft Messenger from the Microsoft corperation.
Microsoft Messenger holds buddies in a list and shows for each user in the list his current status to indicate whether he is *Online*, *Offline*, *Away* etc. For better administration users could be divided into groups. An audio conversation starts easily by clicking on a button in the Messenger window on the selected user to call. When a user receives a message, the Messenger window is opened automatically indicating the person who has send it. Many people are using now live audio conversation to avoid the high billing invoices of calling someone in other countries and that is why it lead to very great success.
The subsequent figure 4.7 shows the Microsoft Messenger. The left hand side illustrates the list holding the different buddies and the right hand side shows a messenger window opened for user “Samar”.



Übermitteln Sie bei Sofortnachrichten niemals Ihr Kennwort oder Ihre Kreditkartennummer.

Figure 4.7: Microsoft Messenger

4.8 Java SIP Packages for Mobile Devices

There exist only two Java SIP packages for the J2ME platforms which are Java API for Integrated Networks (JAIN) Sip Lite from the National Institute of Standards and Technology (NIST) and SIP API for J2ME from Nokia. JAIN Sip Lite is open source. The SIP API for J2ME could only be used for educational use for free and for a limited time only. The main problems in design of applications for mobile devices is the small memory requirements, therefore attention is required to ensure a small memory footprint.

4.8.1 JAIN Sip Lite from National Institute of Standards and Technoloy

JAIN stands for Java API for Integrated Networks, in the INAP (Intelligent Network Application Protocol) specification. It is an interface for wireless devices that provides a uniform interface to wireless, traditional Internet access, the Public Switched Telephone Network (PSTN), and Asynchronous Transfer Mode (ATM).
JAIN SIP Lite is a high-level API. Developers can easily create applications, such as User Agents, which have SIP as their underlying protocol. JAIN SIP Lite is a thin Java API that can be used as a high-level wrapper around the SIP protocol that will support application developers with an easy to use API.

The target audience for JAIN SIP Lite are application developers. In writing applications, the complexity of SIP is of no real relevance. Therefore, the developer will not need to know or be concerned with any of the protocol details. For this reason, the vendor-specific implementation of the API will be responsible for headers such as *Call-ID*, *CSeq*, and *Via*. Therefore, exposure of such headers are not necessary.

The purpose of the JAIN SIP Lite API will be to support SIP functionality as defined in the old RFC SIP specification **[RFC2543]**. This implies that any application that uses this API should be able to perform UA functionality as defined in this old SIP specification.

Architecture of JAIN SIP Lite

The basic model behind JAIN SIP Lite is that of a three-tier approach which consists of the three main classes *CallProvider*, *Call* and *Dialog*. The *CallProvider* has two roles. In a typical JAIN architecture, this class is acting as the Provider. The role of the *CallProvider* is to create *Calls*. A *Call* is the second level of the architecture. Each *Call* created is identified by a *call-id* which remains constant over the lifetime of the *Call*. The *Call* is responsible for creation and management of *Dialogs*. The *Dialog* is the third level of the architecture. Each *Dialog* is created using a *To* address and a *From* address which remain constant over the lifetime of the *Dialog*. The *Dialog* also carries the *call-id* of the *Call* that created it. This also remains contant over the lifetime of the *Dialog*. The *Dialog* is responsible for message creation and management. Configuration and initial set-up is the responsibility of the *SIPStack* object which provides an entry point into the stack and access to the *CallProvider* object. None of the *Listener* interfaces of Figureb4.8 below implement the *java.util.EventListener* to ensure compliance with J2ME.

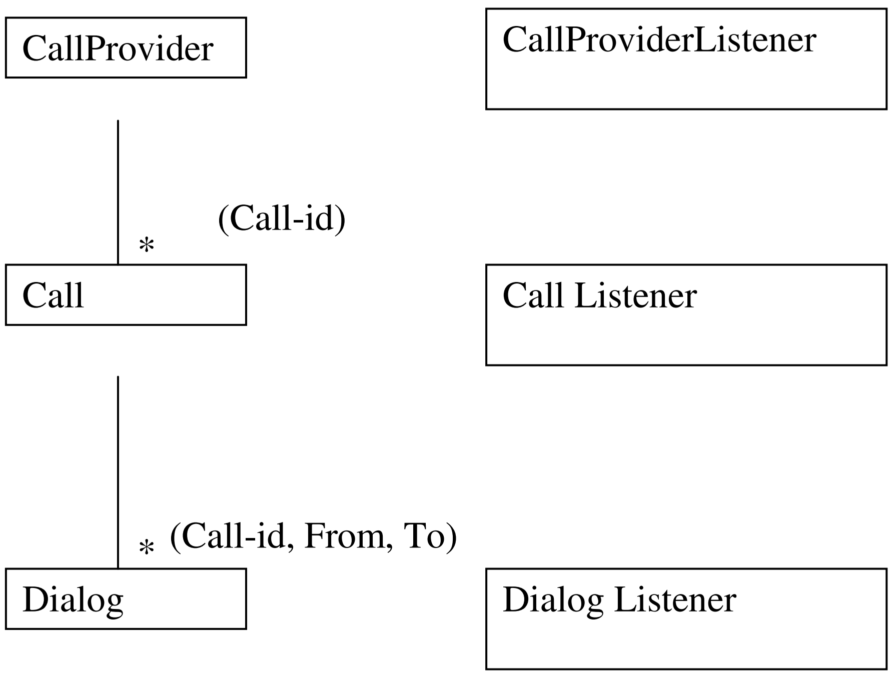


Figure 4.8 Jain SIP Lite architecture

Call

The *Call* object is responsible for creating and managing *Dialogs*. On creation, a *Call* object is created with a unique *call-id* which remains constant over the lifetime of the *Call*. All *Dialogs* created by a *Call* object will contain the same *call-id*. In this way *Dialogs* can be connected to a specific *Call*. For incoming requests and responses, the *Dialog* can be retrieved using this incoming message.

CallListener

The *CallListener* interface listens for all new incoming *Dialogs*. The *CallListener* interface is implemented by the application so that all new *Dialogs* within a call can be handled as neccessary. A *CallListener* implementation must be registered with the *Call* in order that events can be recieved by the listening application.

Dialog

The *Dialog* object is responsible for creation and management of both, request and response messages. Each *Dialog* is created with a *To* address and a *From* address which remain constant over the lifetime of the *Dialog*. The *Dialog* is also created with the unique *call-id* from the *Call* object which is responisble for its creation.

DialogListener

The *DialogListener* interface listens for all incoming requests and responses for the specific *Dialog* it is registered with. Incoming messages are routed through the appropriate method, depending on whether they are requests or responses. A *DialogListener* implementation must be registered with the *Dialog* in order that incoming messages can be recieved by the listening application.

4.8.2 SIP API for J2ME from Nokia

SIP API for J2ME is a Java API written from Nokia. The implementation of the SIP for J2ME API will be integrated in the mobile phone and will make the link between the terminal's native SIP implementation and the the Mobile Information Device Profile (MIDP), environment.
As more and more small devices are supporting the J2ME platform, it is essential that any specification targeting that domain extends the Generic Connection Framework (GCF) pattern to integrate easily with that platform. This means, every new SIP connection can be obtained through the unique connector factory. SIP for J2ME also follows the simple and lightweight structure that all the other protocol frameworks standardized in MIDP do, i.e. *HttpConnection*, *SocketConnection*. This ensures a very flat class structure that inherently simplifies usage.

Similar to *HttpConnection* in the J2ME platform, SIP for J2ME is defined at the transaction level. This choice makes the API multipurpose and does not limit its use by including any assumptions of its intended usage, e.g. Voice over IP.

As a consequence, a MIDlet that is implemented at the transaction level must handle the flow of messages. A MIDlet is a programm written in Java and runs on mobile phones as explained in Chapter 3. The Hypertext Transfer Protocol (HTTP) like functionality has been extended to support the receiving of requests that exist in SIP and HTTP, e.g. blocking calls are extended with an event mechanism that allow application developers to choose the optimal programming style.

For the sake of reducing the MIDlet code size, some helper functions are provided to assist in the basic SIP tasks **[SUNSIP]**. Note, that a MIDlet is free to ignore these helper functions to make previous flexibility requirements possible. Examples of helper functions include:

- Automatic initialization of mandatory headers in requests
- Creation of pre-initialized responses and acknowledgments
- Support for subsequent SIP dialogs
- Support for automatic request refreshes

Architecture of SIP API for J2ME

The API is designed to be a compact and generic SIP API, which provides SIP functionality at transaction level. The API is integrated in the GCF defined in the Connected Limited Device Configuration (CLDC). The SIP API for J2ME is designed as an optional package that can be used with many J2ME profiles. Figure 4.9 shows the simplified class diagram of the API, relation of classes, inheritance from *javax.microedition.Connection* and relation to *javax.microedition.Connector.*

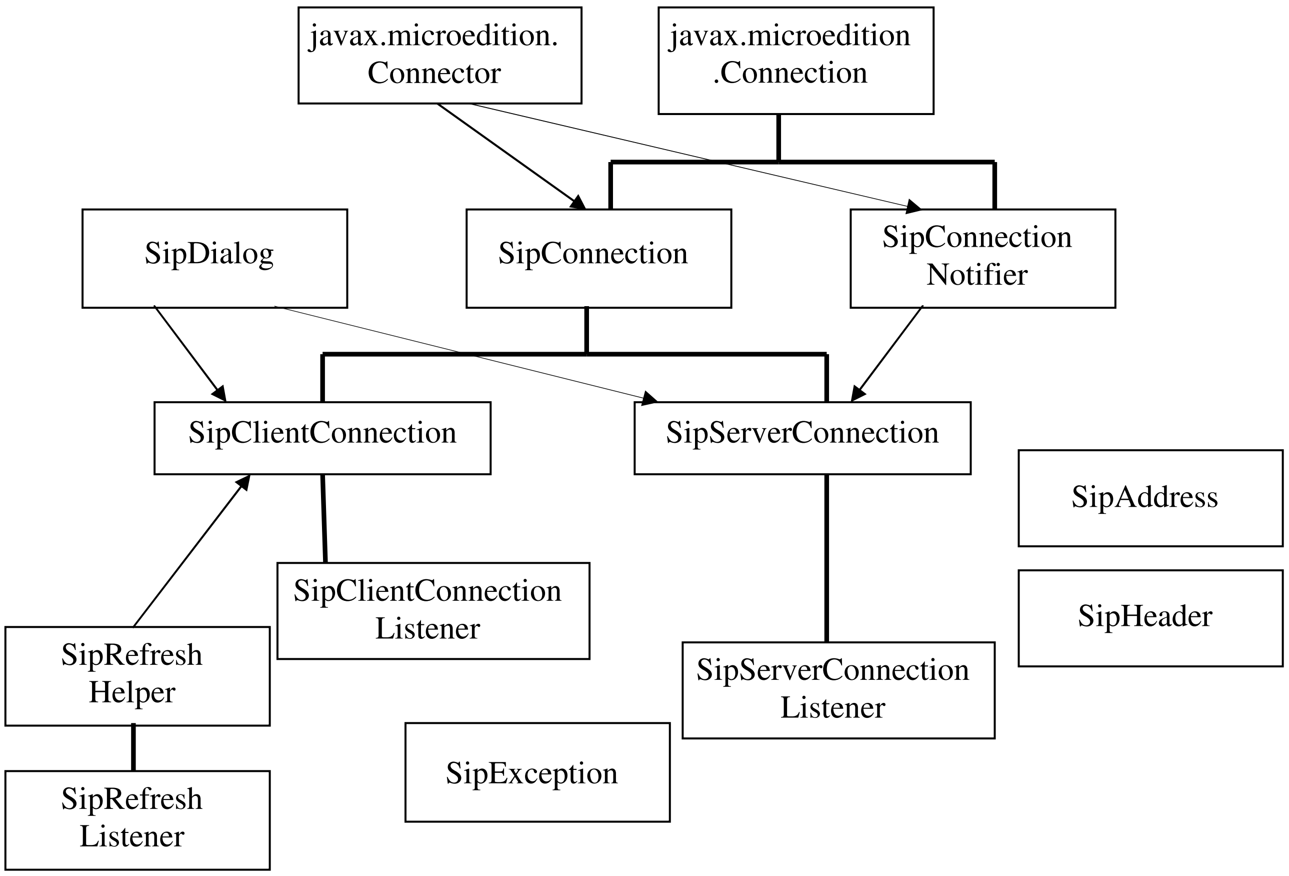


Figure 4.9 SIP API for J2ME architecture

The subsequent figure shows what interfaces a terminal is using to implement SIP User Agent Client (UAC) and User Agent Server (UAS) functionality respectively with SIP API for J2ME. In reality, applications will use both, SIP client and server connections in the same terminal (terminal A in Figure 4.10) and thus implementing both, UAC and UAS functionality **[NOKIA]**.

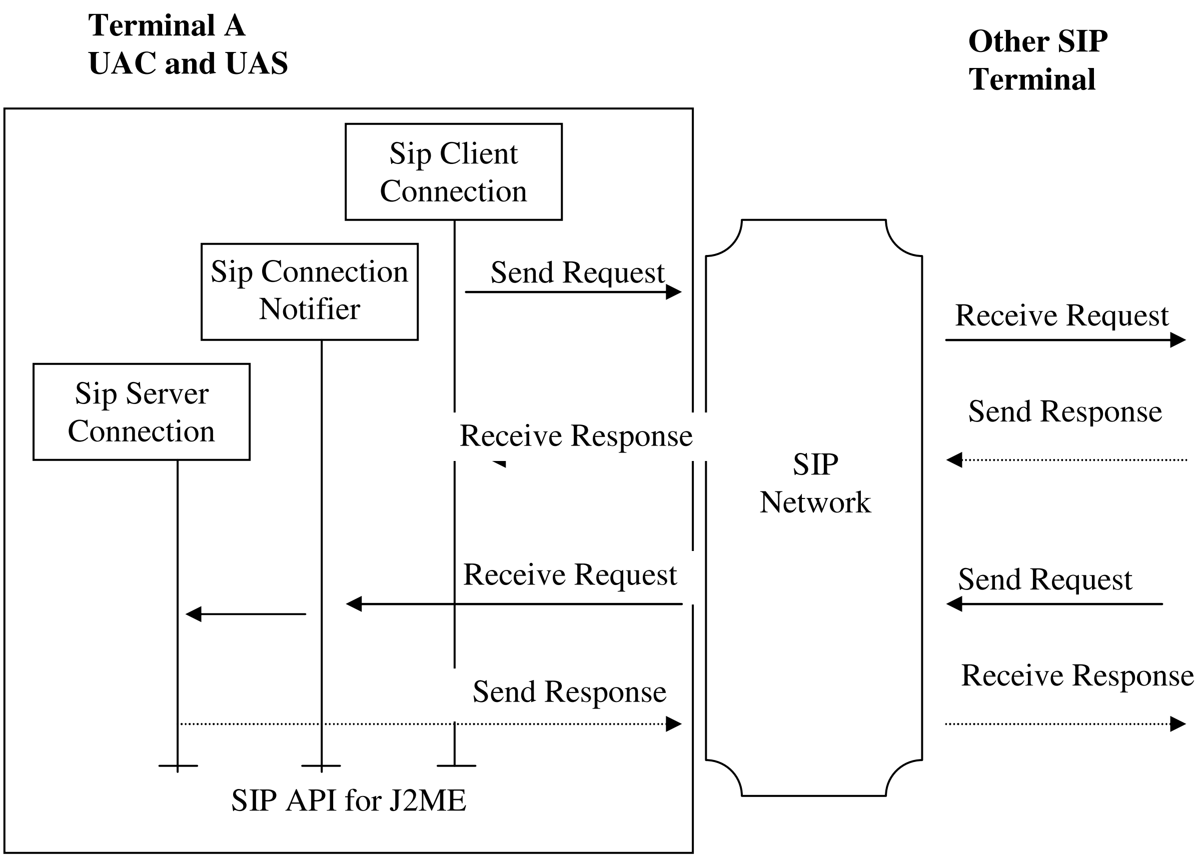


Figure 4.10 SIP API for J2ME functionality

4.8.3 Comparison of SIP Packages

As explained SIP for J2ME from Nokia is part of the MIDP which leads to the requirement that the mobile phone has the version of MDIP which already includes SIP for J2ME installed on the mobile. That is why the package size and the memory footprint is hold small and perfectly for using in developing applications for mobile devices.
JAIN SIP Lite is a high-level open source API which hides all the detail of the protocol from the developer, but developers are at the same time free to change code parts for improvement on that API which gives the developer a great area of flexebility in development.

	Nokia	NIST
Flexibility in use	- not allowed and not possible to change any code parts	+ open sourse, developer can change any code parts
Memory	+ Not much memory	Needs much memory
Requirements	- Mobile must have J2ME package installed as part of MIDP	+ Flexibel to use on any mobile

Table 4.5 Comparison of SIP packages

4.8.4 The Choice

Although SIP API for J2ME uses a small footprint and is part of the GCF, I decided to use JAIN Sip Lite, because it is open source and optmization is allowed. Moreover, because SIP API for J2ME is part of the CLDC, the mobile should have preinstalled this package included with the SIP API for J2ME and there is unfortantely no device which currently supports this. Nokia emulators could be used for tests but for developing the audio UA, the emulator should also be able to provide audio capturing, which is still not possible.

4.9 Summary

Session Initiation Protocol (SIP) is a protocol, which is responsible for creating, managing and tearing down sessions between one or more end-points in an IP network. There are two main components of SIP system mentioned in this section. The Uas, which act as the end-points, are responsible for creating, sending and responding to SIP requests. Proxies provide name resolution and user location for these SIP requests. The ability to create sessions means that a number of different services, such as Instant Messaging and voice applications become possible. In this section, I explained the SIP protocol and gave some call flow examples to illustrate how components work together. At the end, I showed and explained the Java available SIP packages, a comparison between them and why I have chosen to use the NIST JAIN SIP Lite package.

5 Real-time Transport Protocol

5.1 Introduction

Real-time Transport Protocol (RTP) is a standard transport protocol for transmitting real time data such as interactive audio and video suitable for applications over packet-oriented data networks such as the Internet **[RTP]**.
The mode of transmission can be unicast or multicast type and there is no guarantee of any Quality of Service (QoS) for the real time service. The data transport is monitored by another protocol called Real-time Transport Control protocol (RTCP), which allows monitoring of the data delivery and control of data loss in a manner scalable to large multicast networks.
RTP is proposed by the Internet Engineering Task Force (IETF) in the Request for Comments (RFC) 1889. RTP is accepted as a universal standard for the real time multimedia transmission.

5.2 How does RTP work?

As an example to show how RTP is working, a simple multicast audio conference is explained. In this Thesis I am concentrating on audio transmission over RTP for mobile devices.
Participants in an audio conference are sending their audio data in small data chunks of say 20 ms duration. Each chunk is preceded by a RTP header and that is again put into an User Datagram Protocol (UDP) packet **[RFC768]**. The Transmission Control Protocol (TCP) cannot support the real time services because the fact that TCP is rather a slow protocol, requiring a three way hand shake **[RFC793]**. Hence, UDP is used as transport protocol. Although UDP is an unreliable protocol, which does not support retransmissions upon packet loss, it has some features like multiplexing and check sum services, which favors the real time services. RTP has some various services to solve the problem of lost and double packets. Figure 5.1 illustrates the RTP Stack.

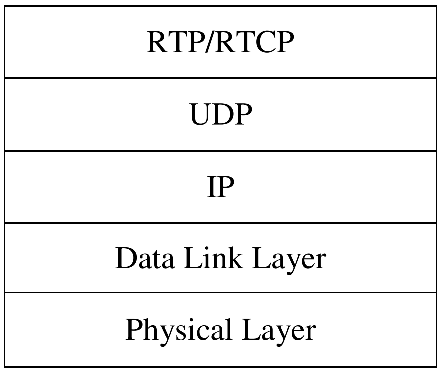
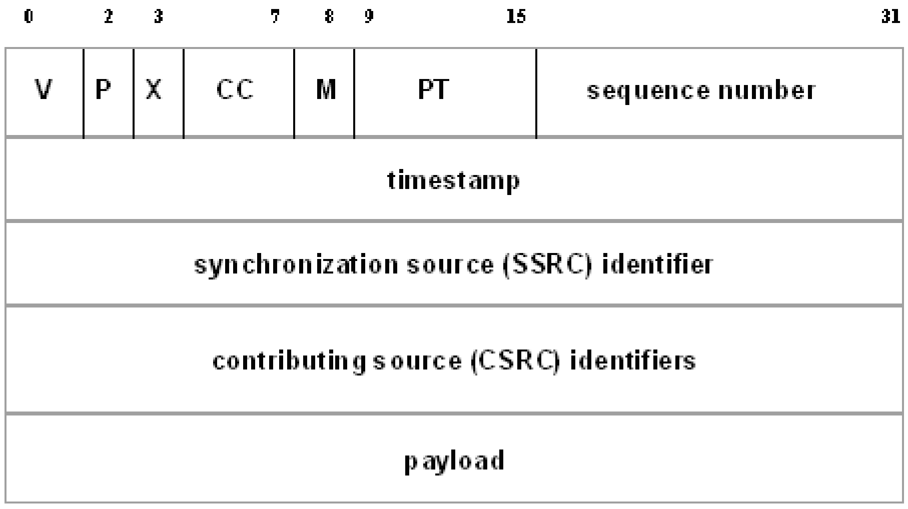


Figure 5.1 Real-time Transport (Control) Protocol Stack

The RTP header contains meta information about the packet and indicates the codec of the data packetd transported, such as Pulse Code Modulation (PCM) or Adaptive Differential Pulse Code Modulation (ADPCM). This is important to let the other side when receiving the packet knows which decoding to use to return the data in its original format. The RTP packet could reach the receiving side in a wrong order, so a sequence number and timing info are also included in the RTP header to let the receiver be able to reconstruct the packets order. The sequence number is also important to calculate how many packets are being lost during transmission. The audio or video application periodically multicasts a reception report with the name of its user on the RTCP port. The report contains information about the media stream, it shows for example, how many packets have been lost. It informs the RTP layer to adjust its coding and transmission parameters for the proper delivery of the data. When a participant wants to leave the conference, it sends a RTCP BYE packet. Audio and video data are transmitted as separate RTP sessions and RTCP packets are transmitted for each medium using two UDP port pairs and or multicast addresses.

5.3 RTP Header Fields

The RTP header has the following format. The first twelve octets are present in every RTP packet, while the list of Contributing source (CSRC) identifiers are present only in packets, which are passed. A CSRC is a source of a stream of RTP packets that has contributed to the combined stream produced by an RTP mixer. A mixer receives streams of RTP data packets from one or more sources, possibly changes the data format, combines the stream in some manner and then forward the combined stream adding its own mixer's identifier called Synchronization source (SSRC*)* in order to preserve the identity of the original sources contributing to the mixed packet. Susequently, I outline the responsibilities of each RTP header field illustrated in Figure 5.2.



0
2
3
7
8
9
15
31
V
P
X
CC
M
PT
sequence number
timestamp
synchronization source (SSRC) identifier
contributing source (CSRC) identifiers
payload

Figure 5.2 RTP Header Fields

Version (V): 2 bits , shows the current version of RTP, which is 2.

Padding (P): 1 bit. If the padding bit is set, the packet contains one or more additional padding octets at the end which are not part of the payload. The last octet of the padding contains a count of how many padding octets should be ignored. Padding may be needed by some encryption algorithms with fixed block sizes or for carrying several RTP packets in a lower-layer protocol data.

Extension (X): 1 bit. If X is set, the fixed header is followed by exactly one header extension.

CSRC count (CC): 4 bits. The field indicates the number of CSRC identifiers that follow the fixed headers. As mentioned before, the field has a non-zero value only if passed through a mixer.

Marker bit (M): 1 bit. If M is set, it indicates some significant events like frame boundaries to be marked in the packet stream. For example, an RTP marker bit is set if the packet contains a few bits of the previous frame along with the current frame.

Payload type (PT): 7 bits. PT indicates the payload type carried by the RTP packet. RTP Audio Video Profile (AVP) contains a default static mapping of payload type codes to payload formats.

Sequence number: 16 bits. The number increments by one for each RTP data packet sent, with the initial value set to a random value. The receiver can use the sequence number not only to detect packet loss but also to restore the packet sequence.

Time stamp: 32 bits. The time stamp reflects the sampling instant of the first octet in the RTP data packet. The sampling instant must be derived from a clock that increments monotonically and linearly in time to allow synchronization and jitter calculations at the receiver. The initial value should be random, so as to prevent known plain text attacks. For example, if the RTP source is using a codec, which is buffering 20 ms of audio data, the RTP time stamp must be incremented by 160 for every packet irrespective of the fact that the packet is transmitted or dropped.

SSRC: 32 bits. This field identifies the source that is generating the RTP packets for this session. The identifier is chosen randomly, so that no two sources within the same RTP session have the same value.

CSRC list: The list identifies the contributing sources for the payload contained in this packet. The maximum number of identifiers is limited to 15, as is apparent from the CC field (All zeros is prohibited in CC field). If there are more than 15 contributing sources, only the first fifteen sources are identified.

One observation that can be drawn from the RTP packet is that it does not contain the delimiting field, as it is done in the lower layer protocol data units (PDU). The reason behind this is that the payload of RTP is the same as that of IP payload and hence not required. If the same user is using multiple media during a session, say for example audio and video, separate RTP sessions are opened for each one of them. Hence there is no multiplexing of media at the RTP level. It is up to the lower layers to multiplex the packets from various media and send on a single channel. But RTCP maintains one identifier called CNAME, which is the same for all the media initiated by one user. Hence CNAME is the only identifier at the RTP layer level that can identify the media originated from a user. The overhead of RTP header is considerably large as seen from above. To reduce this, RTP header compression is proposed.

In the prototype of this Thesis, the RTCP is out of scope although it is an inportant part of the RTP protocol. It was not implemented due to the limited performance and memory capability of the mobile devices, so I am not going into more detail of RTCP especially, because it offers control and monitoring, but it never gurantees QoS.

5.4 RTP and the Session Initiation Protocol

The IETF has proposed the Session Initiation Protocol (SIP) for establishing, modifying and terminating multimedia calls over the Internet as mentioned in Section 4 in detail. RTP/RTCP is used as the protocol for media transfer. The detailed flow diagram of SIP was presented in Figure 4.2 of the basic SIP seesion setup.
The caller User Agent Client sends an *INVITE* message to a friend. This message also contains an SDP packet describing the media capabilities of the calling terminal. The UAS or SIP proxy server receives the request and immediately responds with a *100 (Trying)* response message. The UAS starts ringing to inform the callee of the new call. Simultaneously, a *180 (Ringing)* message is sent to the UAC. The friend picks up the call and the UAS sends a *200 (OK)* message to the calling UA. This message also contains an SDP packet describing the media capabilities of the friend's terminal. The calling UAC sends an *ACK* request to confirm the *200 (OK)* response was received. After that an RTP session is opened on the port written in the SDP packet and the Uas start to send rtp packets. An example SDP description is shown in Listing 5.1.

```
v=0
o=UserA 2890844526 2890844526 IN IP4 here.com
s=Session SDP
c=IN IP4 100.101.102.103
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Listing 5.1 SDP Fields

After Samya's UA gets this request the SDP part got parsed. In this example the UA knows after parsing to send RTP packets encoded audio in PCMU 8000Hz on IP Address 100.101.102.103:49172.A UA handling a media flow that comprises several "m" lines sends media to different destinations (IP address/port number) depending on the codec used at any moment. If several "m" lines contain the codec used media is sent to different destinations in parallel as shown in Listing 5.2 .

```
m=audio 30000 RTP/AVP 0
a=rtpmap:0 PCMU/8000
m=audio 30002 RTP/AVP 8
a=rtpmap:0 PCMU/8000
m=audio 30004 RTP/AVP 0 8
a=rtpmap:0 PCMU/8000
```

Listing 5.2 SDP Fields for opening multiple rtp sessions

This would mean if it is part of example in Listing 5.1 that RTP packets encoded audio in PCMU 8000Hz have to be send on IP Address 100.101.102.103:30000, 100.101.102.103:30002 and 100.101.102.103:30004.

5.5 Java Packages for Mobile Multimedia

Java packages for Mobile Multimedia should extends the functionality of the J2ME platform by providing audio, video and other time-based multimedia support to resource-constrained devices. It should allow Java developers to gain access to native multimedia services available on a given device. There is only one Java package for mobile multimedia available on the market, which is the Mobile Media Application Programming Interface (MMAPI).

5.5.1 Mobile Media Application Programming Interface

The MMAPI provides support for multimedia applications on Java-enabled devices. It is a small part of the Java Media Framework (JMF) and developed by the same group and it is the only Java package available for developing multimedia applications. Unfortunately, it has not all the capabilities of the JMF, but it provides the developer with basic functionalities.
The devices, on which the media run, can range from simple cellular phones to more sophisticated devices, such as PDAs and set-top boxes that support advanced sound and multimedia capabilities. This API allows simple access and control of time-based media, such as audio and video, and is both, scalable and extensible to support more sophisticated multimedia features.
This API is an optional package, which means, that the different functionalities supported are not mandatory included in each mobile which has the package preinstalled. The different features could be checked by different functions or explained separately in a vendors specific mobile/device document.

In general, the MMAPI is a Java package that allows a consumer device to access time-based multimedia functionality, such as audio clips, Musical Instrument Digital Interface (MIDI) sequences, movie, clips, and animations. Time-based multimedia plays an important role in many applications. In the MMAPI, the details of which codecs are supported are left to the profile (such as the Mobile Information Device Profile described in Chapter 3.4). Audio formats supported, are only Pulse Code Modulation (PCM) and Wave (WAV) formats. I will go into more detail about the technical part of MMAPI in the subsequent chapter.

6 The Prototype

6.1 Introduction

For development of the SIP User Agent (UA) for mobile devices, I used a text editor and the Wireless Toolkit (WTK) from SUN (chapter 3). To develop this UA, I went through the following steps:

- Construct small MIDlets to get the functional know how of the Graphical User Interface (GUI), the Generic Connection Frameworks (GCF) and the Mobile Media API (MMAPI) package.

- Implementing the Real-time Transport Protocol (RTP) and testing it by sending packets and receiving them, checking if all packets are received in the right order and that no packets are lost.

- Downloading a free SIP proxy Server from Brekeke **[BREKEKE]**(section 6.7) to register to it and to test the prototype. The web based dministration tool helped me to verify who was registered and what calls were in progress. I was able to check the registration of the SIP UA at the SIP server by logging in at the SIP Server.

- Testing the JAIN SIP Lite package and getting used to its functionalities.

- Constructing the prototype by using the JAIN SIP Lite packge and test example with the MMAPI and RTP implementation to get the functionalities needed. The registration of the SIP UA to the proxy was checked by the Ondo SIP server admin web tool.

- Drawing the class diagram, sequence diagram and sketch on papers how the user interface will be with navigation.

6.2 Requirements

The SIP UA consists of three main parts which are JAIN SIP Lite, Mobile Media API and the RTP Protocol package. To be able to let the SIP UA run on a mobile phone, it must contain the following:

- MMAPI must be preinstalled on the mobile and voice capturing must be allowed on this mobile phone

- Connected Limited Device Configuration (CLDC) version 1.0 and Mobile Information Device Profile (MIDP) version 2.0 must also be preinstalled

Mobile phones having those packages preinstalled, have the required memory and Central Processing Unit (CPU) speed suitable for thie UA implementation.
Attention has to be given to the MMAPI, because not every mobile having this package preinstalled has the capability to capture audio. This is an optional property and has to be checked in the manufactory papers for the specific mobile phone on which the UA is intended to be run.

6.3 RTP Implementation

The Real-time Transport Protocol (RTP) has become a widely implemented Internet standard protocol for transport of real-time data. This implementation, which is available as *at.ikn.j2me.protocol.RTP* package, can be easily incorporated into an application, which then has access to all the transport level features that RTP provides.

The overall architecture was determined by the need to have a modular, simple and platform independent RTP implementation to easily integrate into any application requiring RTP. The RTP part is explained seperatly from the SIP part.

Session:
The top most class the user of the package interacts with is *Session*. This class encapsulates the RTP related setup, startup and shutdown procedures. *Session* also serves as the interface where it is possible to control object's states and their interactions.
From the user's perspective, the *Session* interacts with the network and is responsible for sending and receiving RTP packets. Network interaction constitutes can be classified into two distinct processes:

- Synchronous processes: Sending RTP packets
- Asynchronous processes: Receiving RTP packets

The synchronous interaction with the network is straightforward, this is driven by the application (the user of the package) and is invoked by calling the *Session.SendRTPPacket()* function.

The asynchronous interaction, on the other hand, requires the RTP receivers to run on separate threads and wait for the packet arrival. Following the reception of a packet, several tasks are performed among which is posting of the packet arrival event.

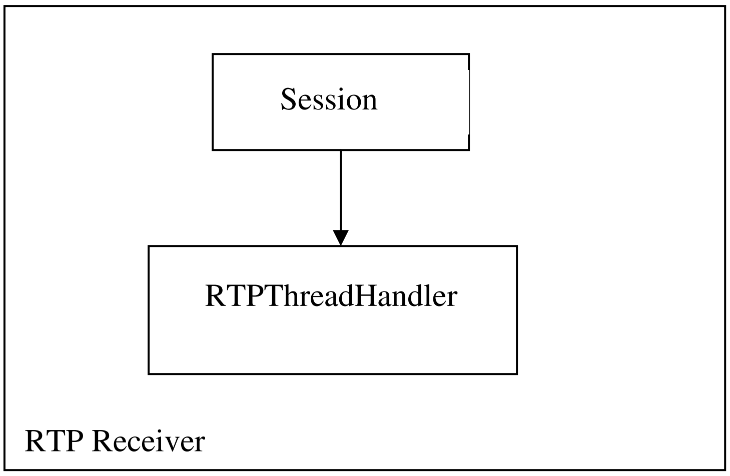


Figure 6.1 High level functional module

RTP Packet Sending and Receiving:
Figure 6.1 shows the object model where *Session* uses the *RTPThreadHandler* object to send and receive RTP packets. The *RTPThreadHandler* class is inherited from *Thread* and the

function which receives the RTP packets, runs in an infinite loop. This loop is started directly by calling or using *Session.Start()* method which starts the Receiver. *RTPThreadHandler* class also acts as a sender by providing the *SendPacket()* function.

RTP reception updates the *Session* information that are necessary to hold an RTP session with minimal interaction from the application. This makes it easy for applications, which do not necessary need to concern themselves with the protocol details, to use this package and with only a few lines of code, have a fully functional RTP session. An example shows Listing 6.1.

```
// Construct a new Session object
rtpSession = new Session ( "234.5.6.7", // ToIPAddress
                                          8000, // localReceivPort
                                         8001, // RTPSendToPort
);

// Set the session parameters
rtpSession.setPayloadType ( 5 );

// Start the session
rtpSession.Start();

// Send a test packet.
rtpSession.SendPacket ( String ( "Test" ).getBytes() );

// Stop the session
rtpSession.Stop();
```

Listing 6.1 Construction of a RTP session

It was mentioned earlier that when a RTP packet arrives, session level states are updated. Furthermore, it is necessary to forward the RTP packets over to the user of the session (application or any other class requiring the received RTP packets).

The event model in this package is based on the one implemented by the Java Abstract Windowing Toolkit (AWT) 1.1. In the AWT event-model, asynchronous events, such as button-click, are handled by registered listeners. A listener is a class, which implements an interface specified by the event-model. For instance, there exists the *mouseListener* interface in AWT. Any class interested in mouse events must implement this interface. Additionally, the interested class instance must register itself with the event source by calling the method *addMouseListener* and passing its own instance reference. This step of registration will allow the event dispatcher to call the appropriate event handling functions in the listener when an event is posted. The dispatcher guarantees that the listener, which declares that it implements the interface, indeed does implement all the methods. This guarantee is in effect given by the compiler, which will produce compile-time errors until an implementation, even if empty, is provided by the interface implementor.
Following the same model, an interface, *RTP_actionListener* is made available. It is not required by the application classes to implement any of these interfaces. The only requirement is that once an application class is declared that it implements the interface, it must provide handlers for each method. The objects passed to the event handler are classes which encapsulate the RTP packets.

The RTP Packet class and all its attributes are illustrated in Figure 6.2.

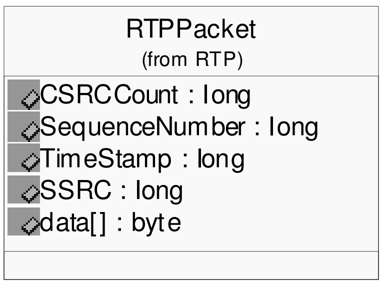


Figure 6.2 RTPPacket class

When the *Session.postAction()* is called, the registered *RTP_actionListener* is checked.. This method is provided with the newly constructed *RTPPacket* object. An alternative is to post the byte data stream instead, but from the user's point of view, it is easier to work with objects and attributes than to parse bits and bytes out of a byte datagram. To the user, an incoming RTP packet is nothing more than an object instance of the *RTPPacket* class. The user is not required to do anything with any of the attributes, most often, it will only need the data attribute, which contains the payload.

The sequence diagram in Figure 6.3 shows the interaction between the *Session*, the session instantiator, which implements the *RTP_actionListener* interface, the RTP receiver and the *RTPPacket* object.

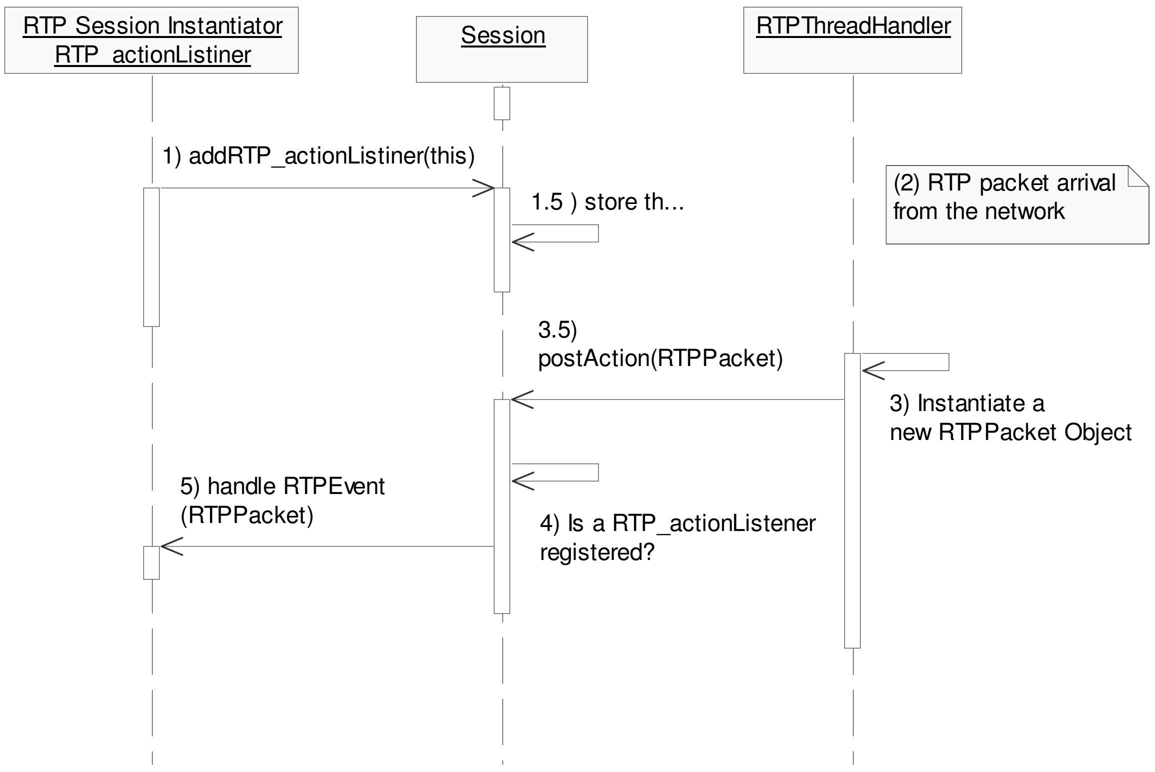

RTP Session Instantiator
RTP_actionListiner
Session
RTPThreadHandler
1) addRTP_actionListiner(this)
1.5) store th...
(2) RTP packet arrival from the network
3.5) postAction(RTPPacket)
3) Instantiate a new RTPPacket Object
5) handle RTPEvent (RTPPacket)
4) Is a RTP_actionListener registered?

Figure 6.3 RTP Sequence Diagram

Sequence 1 and 1.5: The registration takes place. Here, instance of the class which is implementing the reference in order to post the RTP events to this listener.

Sequence 2: The asynchronous event, i.e. the RTP packet reception.

Sequence 3: The RTP receiver, instantiates a *RTPPacket* object and populates its fields. It then posts the packet object to the *Session* in sequence 3.5.

Sequence 4: The *Session* determines if any *RTP_actionListener* is registered. In this case, the Listener waits for a notification.

Sequence 5: The packet object is sent to the *RTP_actionListener*.

In general, following rules and guidelines apply while working with the events model and actionListener interfaces:
A class must implement the *RTP_actionListener* interface:

class MyClass implements RTP_actionListener { …}

Only one registered listener will get the event notifications. For instance, there are two classes, A and B, that implement the *RTP_actionListener* interface, and if instance of A followed by B registers itself by calling the *Session.addRTP_actionListener()* function, then the last registration will overwrite the one preceding it. It is necessary to understand that there can only be one instance registered for *RTP_actionListener*. If a class implements both interfaces, it must provide implementation for methods in both the interfaces.

Code examples in Listing 6.2 may help clarify the event model.

```
// One class that implements the RTP_actionListener interface and registers itself

class MyClass implements RTP_actionListener {

        MyClass {
                Session rtpSession = new Session (…);
                rtpSession.addRTP_actionListener ( this );
                …
        }

        public void handleRTPEvent( RTPPacket rtppkt) {
                // RTP event handler..
        }

} // end
```

Listing 6.2 Class implementing RTP_actionListener interface

RTP Limitations:

Number of RTP Action listener registrations maintained by *java.RTP.Session* is limited to one.

6.4 General Class Architecture

In this section, I explain the general Architecture of the SIP UA. Figure 6.4 shows this architecture. It consist of a main class, which is the UA class.

UA
It does the most important work. After a user presses a certain action in the SignInWindow, the *UA* class decides which class to call. For example, if a user wants to register on a SIP server and waits to receive calls on a certain port, the *UA* calls the *RegisterProcessing* class.

RegisterProcessing
It is responsible for registeration to a SIP server and handles all messages related to this registration.

MessageProcessing
It is responsible for all incomming and outgoing message processing by analysing the content of the incoming message.

ByeProcessing
The *ByeProcessing* class is doing all the work for constructing a *BYE* message or analysing a received *BYE* message.

MicCapture
The *MicCapture* class is responsible for doing the microphone audio capture. It is involved when a deal about the RTP media port is known by the *MessageProcessing* class and the captured samples are put into RTP packets.

PcmPlayer
This class is responsible for a playback of the collected voice samples back.

Session
Is holding a valid session for doing the work of listening to the incoming rtp packets on certain ports.

The *SipUAMidlet*, *MainWindow* and *SignInWindow* are the user interfaces classes.

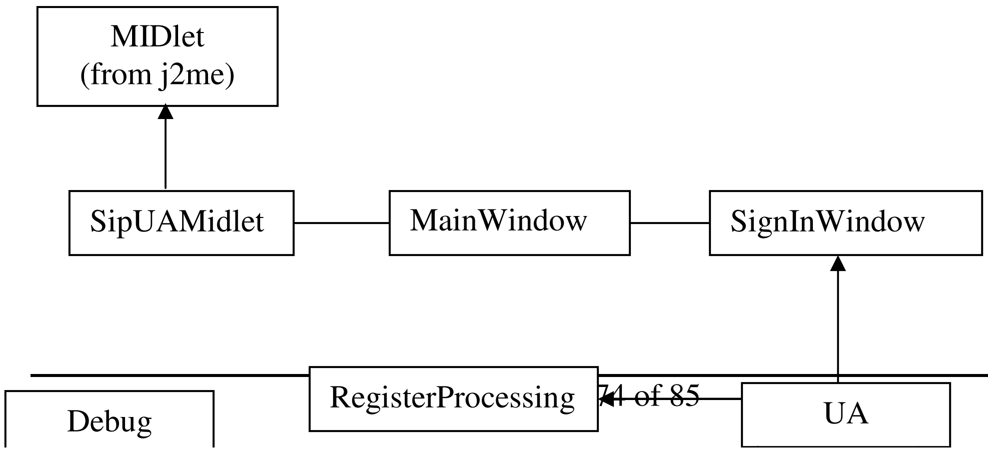
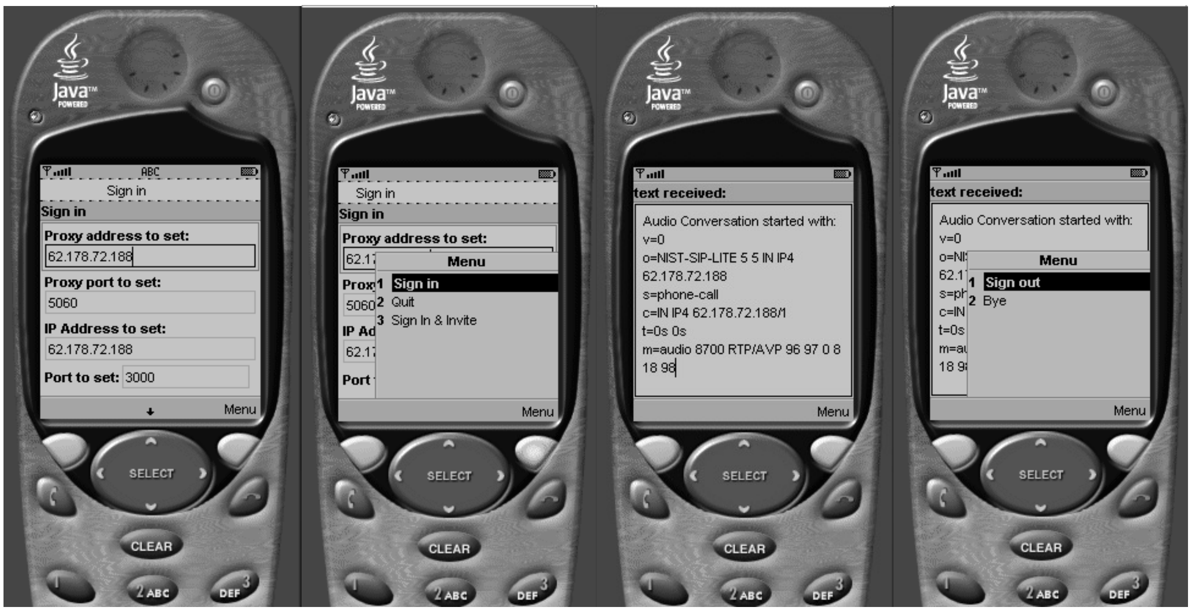


Figure 6.4 General Class Diagram

6.5 Graphical User Interface

When the User Agent is started, a signin window is displayed. The user has to fill in the text boxes for the needed information like the SIP server IP address and port to register there. Its own IP address and the port, on which to receive the calls (RTP packets) and finally the SIP URL and the contact SIP URL. The available menu shows different actions the user can take as *SignIn*, *SignIn&Invite* and *Quit*. If the user chooses to sign in only, it gets registered and waits for any incomming calls. Once an incomming call arrives, a diplay message indicating that the audio conversation has started, is shown. The user can decide any time to either just break down the current call but still listening for other incomming calls or completely sign out and the user will no longer be reachable. This scenario shows Figure 6.5.



Sign in
Proxy address to set: 62.178.72.188
Proxy port to set: 5060
IP Address to set: 62.178.72.188
Port to set: 3000

```
text received:
Audio Conversation started with:
v=0
o=NIST-SIP-LITE 5 5 IN IP4
62.178.72.188
s=phone-call
c=IN IP4 62.178.72.188/1
t=0s 0s
m=audio 8700 RTP/AVP 96 97 0 8
18 98
```

Figure 6.5 Graphical User Interface of the SIP UA

6.6 Mobile Media API

The Mobile Media API package specifies a small multimedia API for Java enabled devices, like simple cellular phones or even more sophisticated, multimedia devices. This API allows simple access and control of audio and video time-based media. It is both, scalable and extensible to support more sophisticated multimedia features.
MMAPI is designed to be protocol and format indifferent. It contains all of the functionality needed to support any new implementation protocols and many more. It allows API implementors and Java profile creators to choose which media formats they will support.

6.6.1 Mobile Media API Architecture

The Mobile Media API is based on four fundamental concepts:

- A P*layer* knows how to interpret media data. One type of player, for example, might know how to produce sound based on MP3 (Motion Picture Experts Group 1 Layer 3) audio data. Another type of player might be capable of showing a QuickTime movie. Players are represented by implementations of the *javax.microedition.media.Player* interface.

- You can use one or more controls to modify the behavior of a *Player*. You can get the controls from a *Player* instance and use them while the *Player* is rendering data from media. For example, you can use a *VolumeControl* to modify the volume of a sampled audio *Player*. Controls are represented by implementations of the *javax.microedition.media.Control* interface; specific control subinterfaces are in the *javax.microedition.media.control* package.

- A data source knows how to get media data from its original location to a *Player*. Media data can be stored in a variety of locations, from remote servers to resource files or Record Management System (RMS) databases. Media data may be transported from its original location to the player using Hypertext Transfer Protocol (HTTP), a streaming protocol like RTP, or some other mechanism. *javax.microedition.media.protocol*.DataSource is the abstract parent class for all data sources in the Mobile Media API.

- Finally, a manager ties everything together and serves as the entry point to the API. The *javax.microedition.media.Manager* class contains static methods for obtaining *Players* or *DataSources* **[MMAPI]**.

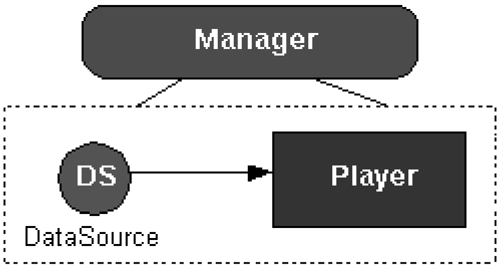


Figure 6.6 Architecture of MMAPI **[SUN]**

The simplest way to obtain a *Player* is to use the first version of *createPlayer()* and pass in a *String* that represents media data. For example, you can specify an audio file on a web server:

Player p = Manager.createPlayer("http://webserver/music.mp3");

There is another *createPlayer()* method, which is used in this Thesis, which allows to create a *Player* from an *InputStream*, because the received RTP packets were collected in a buffer at the receiver side and converted to an *InputStream* and played back using this type of *Player* as shown in Listing6.3.

public static Player createPlayer(InputStream stream, String type) throws IOException, MediaException

Listing 6.3 creatPlayer() Method

When a *Player* is created, the playback begins with the *start()* method. Playback helps to understand the life cycle of a *Player*. Playback consists of four states.
When a *Player* is first created, it is in the *UNREALIZED* state. After a *Player* has located its data, it is in the *REALIZED* state. If a *Player* is rendering an audio file from an HTTP connection to a server, the *Player* reaches *REALIZED* after the HTTP request is sent to the server. The HTTP response is received, and the *DataSource* is ready to begin retrieving audio data. The next state is *PREFETCHED*, and is achieved when the *Player* has read enough data to begin rendering. Finally, when the data is being rendered, the *Player's* state is *STARTED*.
The *Player* interface provides methods for state transitions, both forwards and backwards through the cycle described above. The reason is to provide the application with control over operations that might take a long time. You might, for example, want to push a *Player* through the *REALIZED* and *PREFETCHED* states so that a sound can be played immediately in response to a user action.

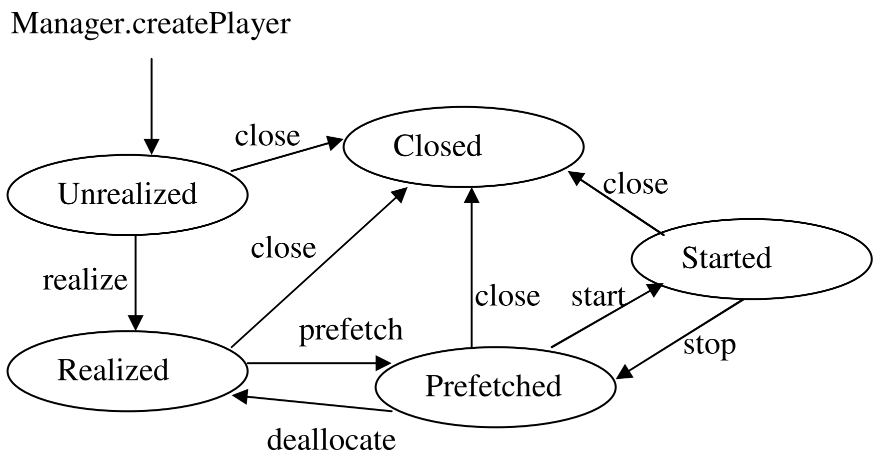


Figure 6.7 Player State Table

The Mobile Media API does not require any specific content types or protocols, but it can find out at runtime what is supported by calling *Manager*'s *getSupportedContentTypes()* and *getSupportedProtocols()* methods. If the *Manager* fails to find a *Player* for a content type or protocol because it is not supported, it will throw an exception. A different content type has to be used in that case or a polite message should be displayed to the user.

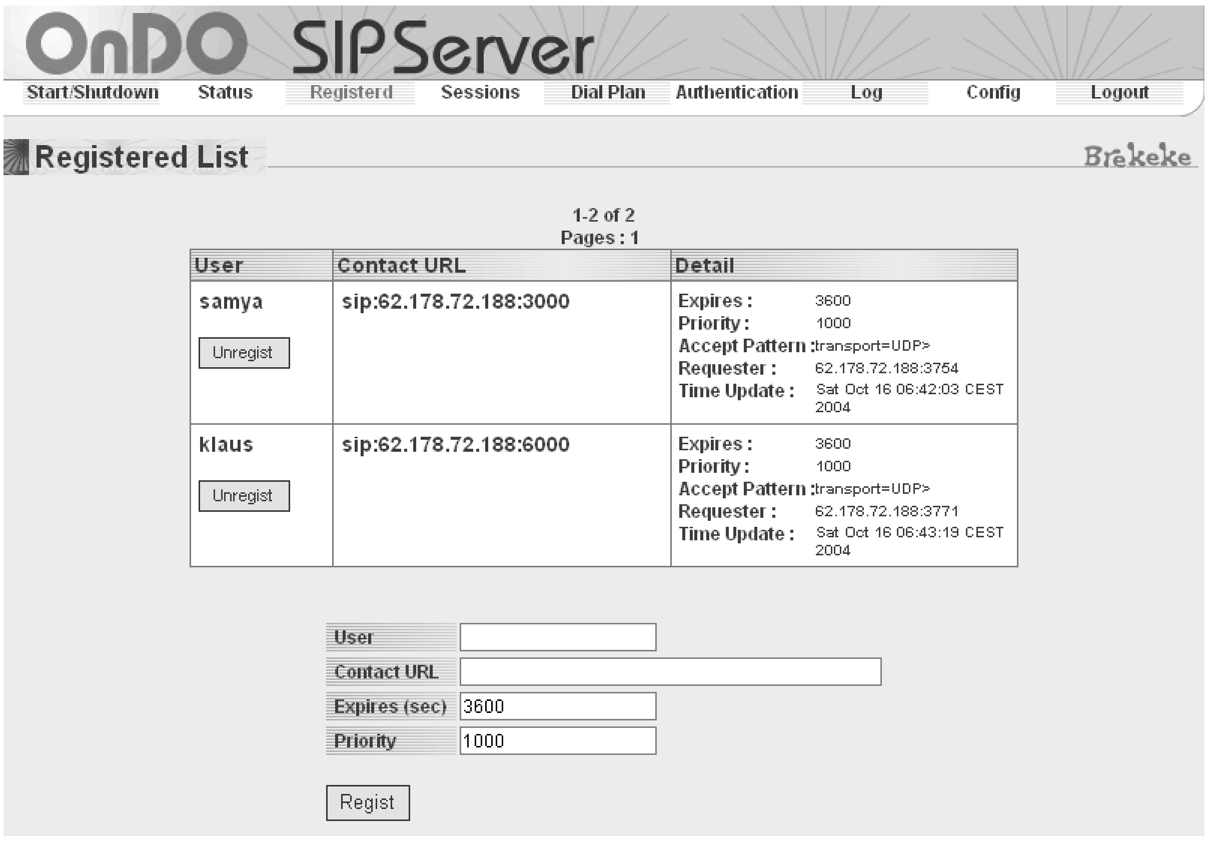
6.7 OnDO SIP Sever from Brekeke

A SIP Servers can act as a SIP registrar, proxy or redirect server as explained in Chapter 4.
The OnDO SIP server from Brekeke is the name of the SIP proxy and registrar server developed and sold by Brekeke Software, Inc. The product has flexible control routing functions.
All operations are performed from the administration tool. Since OnDO SIP Server's administration tool is web-based, maintenance can be performed remotely. From the administration tool, you can start/stop the server, define or modify dial plans, create users or modify settings. When using OnDO SIP Server in educational institutions, licenses are free to students or staff members and could be easily download and installed.

The OnDO SIP Server is a call control server compliant with IETF SIP standard **[RFC3261]**. The server can act as registrar and proxy server, and performs call routing. With OnDO SIP Server, it is possible to use SIP hardphones and SIP softphones.
I was able to verify the registration of the SIP UA at the SIP server by logging in at the SIP server and clicking the *Registered* heading.

The subsequent images show the user interface of the web administration tool:

First a login window appears and by typing the right administrator user name and password, the status of the Sip Server is shown. This could be either *Active* or *Inactive*. By a button press a *Shutdown* of the SIP server can easily be done.
When pressing on the *Registered* link, the SIP server shows all the registered user with detail information, as illustrated in Figure 6.8. The users in the Figure were registered through the prototype and not through the web admin tool.



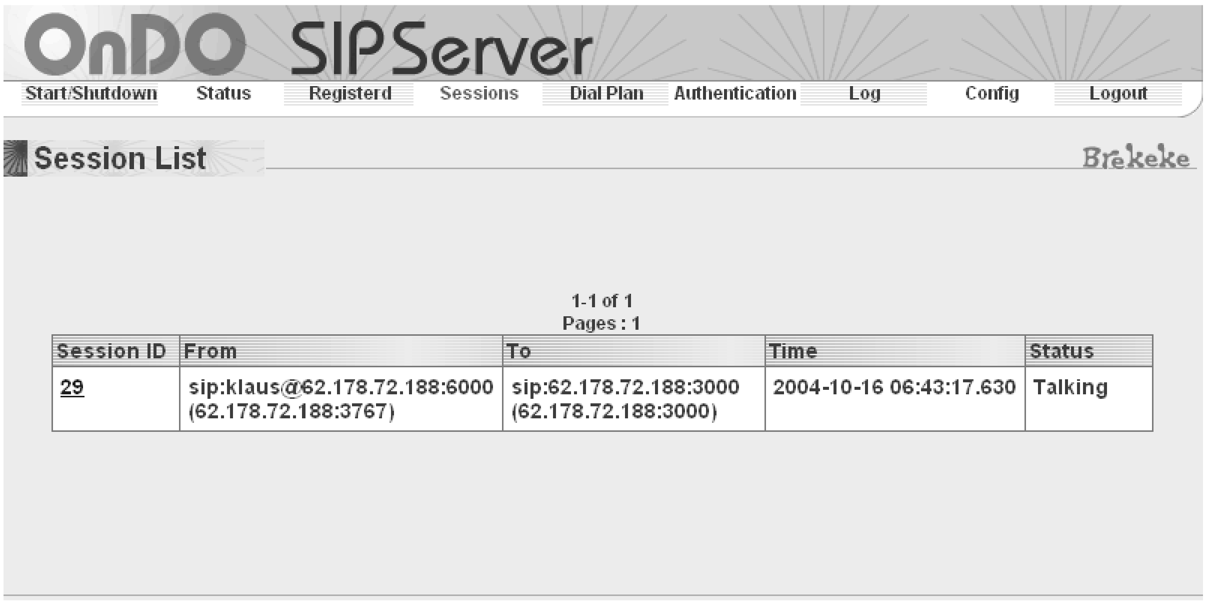
1-2 of 2
Pages : 1

User	Contact URL	Detail
samya	sip:62.178.72.188:3000	Expires : 3600 Priority : 1000 Accept Pattern : transport=UDP> Requester : 62.178.72.188:3754 Time Update : Sat Oct 16 06:42:03 CEST 2004
klaus	sip:62.178.72.188:6000	Expires : 3600 Priority : 1000 Accept Pattern : transport=UDP> Requester : 62.178.72.188:3771 Time Update : Sat Oct 16 06:43:19 CEST 2004

User	
Contact URL	
Expires (sec)	3600
Priority	1000

Figure 6.8 Registered users

The SIP server is able to display the currently running sessions. By clicking on the *Sessions* link and by clicking on the *SessioID*, the detail information about the chosen session is displayed as shown in Figure 6.9.

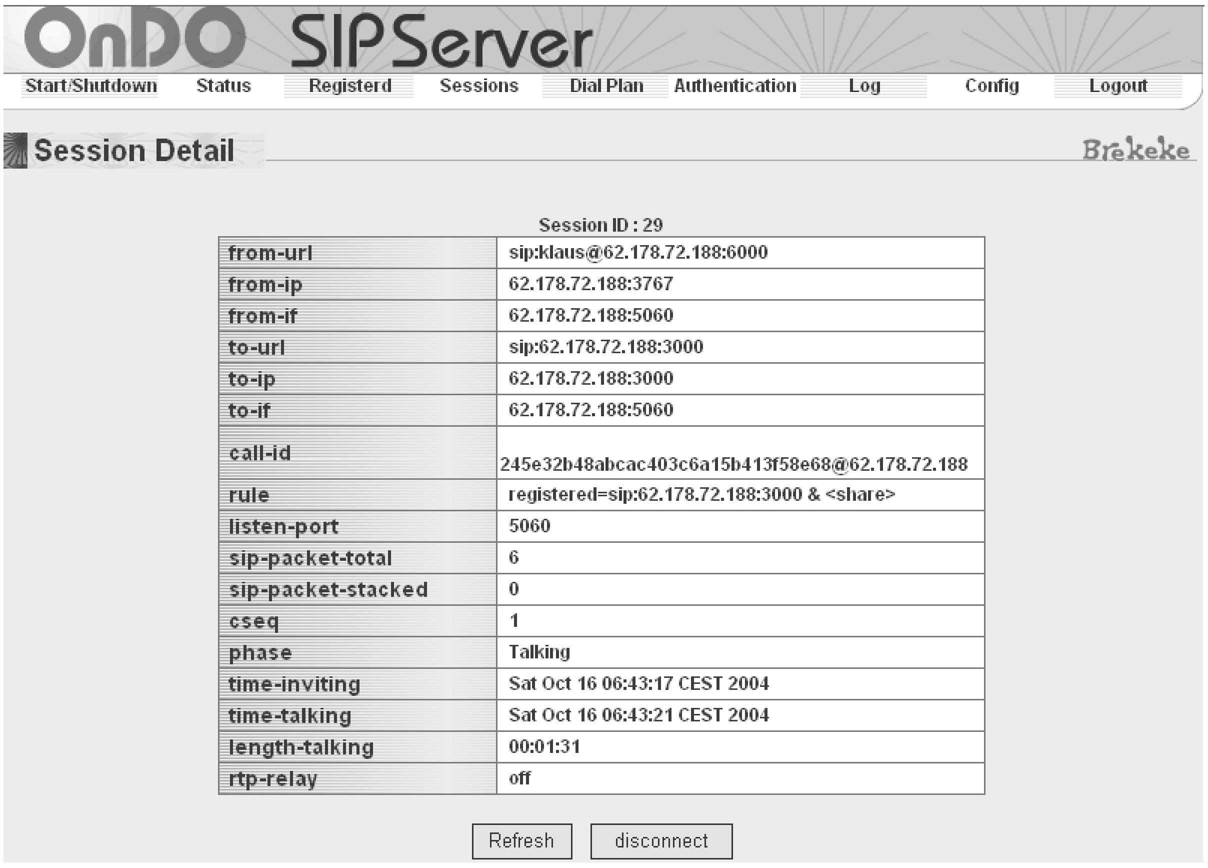


Session List

1-1 of 1
Pages : 1

Session ID	From	To	Time	Status
29	sip:klaus@62.178.72.188:6000 (62.178.72.188:3767)	sip:62.178.72.188:3000 (62.178.72.188:3000)	2004-10-16 06:43:17.630	Talking

Figure 6.9 Active session



Session Detail

Session ID : 29

from-url	sip:klaus@62.178.72.188:6000
from-ip	62.178.72.188:3767
from-if	62.178.72.188:5060
to-url	sip:62.178.72.188:3000
to-ip	62.178.72.188:3000
to-if	62.178.72.188:5060
call-id	245e32b48abcac403c6a15b413f58e68@62.178.72.188
rule	registered=sip:62.178.72.188:3000 & <share>
listen-port	5060
sip-packet-total	6
sip-packet-stacked	0
cseq	1
phase	Talking
time-inviting	Sat Oct 16 06:43:17 CEST 2004
time-talking	Sat Oct 16 06:43:21 CEST 2004
length-talking	00:01:31
rtp-relay	off

Figure 6.12 Detail information about the session

6.8 Audio Playback

Unfortantely, mobile devices are still not powerful enough for playing back audio sampled data. The available player of the MMAPI package needs much time to set the pointer at the beginning of the buffer for every new collected audio sample. This time delay has lead to the problem packets are not played back and therefore no clear audio conversation is possible. Even by trying to use two parallel buffers, the problem is even much worse during player reset, because of the time to switch between the two different buffers. I am sure that mobile phones will get much powerful in the future and a solution like using two different buffers will work.

7 Summary

The aim of this Thesis was to realise a SIP User Agent for mobile devices. The architecture consists of three main parts, which are the SIP, RTP and media playback part. To be platform independent, Java was chosen for the implementation.
First, I gave an introduction about the SIP and the RTP and the motivation for the work presented in this Thesis. Then I discussed the reasons for choosing SIP as the signaling protocol and mentioned a short description of the main topics and contributions of this work. Afterwards, I explained the currently available mobile devices on the market, a comparison between them and a short description about mobile applications. Then, I went deeply through the J2ME framework. I showed the functionality through simple examples and compared the existing Java Virtual Machines and the main packages used for mobile phones. These are the Connected Limited Device Configuration and Mobile Information Device Profile, which have been explained in more detail. How all J2ME components fit together, how network connections can be estabished and the lifecycle of MIDlets have been shown. Finally, the wireless toolkit was introduced, because it was the development tool used in this Thesis. Then in Chapter 4, which is a very important part of this Thesis, I covered the SIP. It started with the definition and showed call flow examples. The SIP messages, requests and responses, were explained. The SIP components were presented and the SDP was introduced. An example of a SIP User Agent with its functionality currently available on the market was shown. Finally, the available Java SIP packages for mobile devices were discussed, explaining the architecture of each with a comparison. These are JAIN SIP Lite and SIP API for J2ME. I chose JAIN SIP Lite for the practical part of this Thesis, because it is open source and optimization is allowed. After that, I gave an overview about the RTP and why it is so important for SIP in the real-time data exchange. The RTP header fields were explained in detail and how RTP fits in the architecture. Finally, the available Java packages for mobile multimedia were intoduced. Chapter 6, which is the final Chapter, includes the practical part of the Thesis, which is the prototype. Starting with the development steps used and the requirements needed to make this implementation possible. This Chapter showed and explained how all components work together. Own implemented parts were shown by sequence and class diagrams. A general class architecture was presented and screenshots of the prototype illustrates how it is really working. Through the Mobile Media API, it was possible to send and play back voice data. The UA was tested and call establishement was verified by the prototype.

8 Conclusion and Outlook

Session Initiation Protocol (SIP) is used for setting up communication sessions such as conferencing, telephony, whiteboarding and instant messaging on the Internet. It bridges the gap between the Internet and conventional telephony. Through the prototype of this Thesis this was verified and wireless IP calls were successfully established.
Commercial SIP-based products and services already include IP phones, PC clients, SIP servers and IP telephony gateways. It has huge potential for use in third-generation (3G) wireless networks, in mobile applications, and in providing the essential infrastructure for Internet telephony, including quality of service and security. So there is a huge future for developing new applications waiting. As stated by the telecomm magazin in march 2003:
Two of the fastest-growing areas of development in the telecom industry in recent years have been mobile telephony and IP-based communications. It's no coincidence, therefore, that these two areas are converging and at the center of this move towards greater mobility for IP is SIP

(Session Initiation Protocol). SIP is becoming for person-to-person IP communications what HTTP is for the Internet and, while much of its early development was focused on fixed-line web services, the attention of many developers has now switched toward mobile networks. Making SIP work with wireless devices will allow next-generation mobile users to access multi-function IP-based services which can combine voice, messaging and e-mail on their cellular handset or PDA. SIP has won some heavyweight backing in the mobile industry; the 3GPP (3G Partnership Project), the body which produces 3G standards for evolved GSM networks, has already decided to base 3G mobile call set-up on SIP. Companies such as Microsoft, Nokia and Vodafone are other notable names to take up the protocol.
SIP addresses the needs of IP telephony from an Internet perspective. IP telephony will replace traditional telephony systems, which includes next generation mobile communication networks. The bandwidth will increase and by greater bandwidth, a very good speech quality and powerful applications will be enabled. SIP will be embedded in 3G phone handsets and Personal Digital Assistants and as shown from the prototype, it is possible to have wireless IP voice sessions on mobile devices.

9 References

[BREKEKE] Brekeke: OnDo Sip Server; http://www.brekeke.com/

[IBMNETW] Networking with J2ME, Entigo, http://www-106.ibm.com/developerworks/wireless/library/wi-jio/ (01 Sep 2002)

[J2ME] Eric Giguère (2002): Java 2 Mirco Edition; Wiley, *Professional Developer's Guide Series*; ISBN 0-471-39065-8

[J2MICRO] James P. White and David A. Hemphill; *Java 2 Micro Edition* - 04/04/2002

[MIDP] C. Enrique Ortiz, Eric Giguère (2001): *Mobile Information Device Profile for Java 2 Mirco Edition*; Wiley, Professional Developer's Guide Series; ISBN 0-471-03465-7

[MMAPI] Eidenberger, Horst : Medienverarbeitung in Java : *Audio und Video mit Java Media Framework & Mobile Media API* / Horst Eidenberger ; Roman Divotkey. - 1. Aufl. . - Heidelberg : dpunkt, 2004.

[NOKIA] Nokia Forum, http://www.forum.nokia.com/

[OR] Kim Topley (2002): *J2ME in a Nutshell*; O'Reilly; ISBN 0-596-00253-X

[RFC3261] J. Rosenberg,H. Schulzrinne,G. Camarillo,A. Johnston,J. Peterson, R. Sparks,M. Handley,E. Schooler, *SIP: Session Initiation Protocol*, Internet Engineering Task Force. RFC 3261, June 2002

[RFC793] Information Sciences Institute, University of Southern California,4676 Admiralty Way, Marina del Rey, California 90291, *Transmission Control Protocol*, Internet Engineering Task Force. RFC 793, September 1981

[RFC2794] P. Calhoun, C. Perkins, *Mobile IP Network Access Identifier Extension for IPv4*, Internet Engineering Task Force. RFC 2794, March 2000

[RFC1924] R. Elz, A Compact Representation of IPv6 Addresses, Internet Engineering Task Force. RFC 1924, April 1996

[RFC768] J. Postel, *User Datagram Protocol*, Internet Engineering Task Force. RFC 768, August 1980

[RIGGS] Roger Riggs u.a. (2001): *Programming Wireless Devices with the Java 2 Platform*, Micro Edition; Addison-Wesley; ISBN 0-201-74627-1

[RTP] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson: *Realtime Transport Protocol*. Internet Engineering Task Force. RFC 1889, Jänner 1996

[SDP] M. Handley, V. Jacobson: *Session Description Protocol*. Internet Engineering Task Force. RFC 2327, April 1998

[SIPOLD] M. Handley, H. Schulzrinne, E. Schooler, J. Rosenberg: *SIP: Session Initiation Protocol*. Internet Engineering Task Force. RFC 2543, März 1999

[SIPTEL] R. Pailer, S. Bessler, K. Peterbauer, V. Nisanyan, J. Stadler: *A Serviceplatform for Internet-Telecom Services using SIP. SmartNet2000*, Mai 2000

[SUN] SUN Mircosystems: Java 2 Micro Edition; http://java.sun.com/j2me/ und SUN Wireless Developer Homepage; http://wireless.java.sun.com/