Stream and system management for networked audio devices

André Eisenmann
André Eisenmann

Stream and system management for networked audio devices

Master’s Thesis

Computer Science

2008
## Contents

1 Introduction 1
   1.1 Context .................................................. 1
   1.2 Problem ..................................................... 2
   1.3 Objectives .................................................. 2
   1.4 Constraints ............................................... 3
   1.5 Outline ..................................................... 3

2 Theoretical background 5
   2.1 Embedded systems ........................................... 5
   2.2 Management software ...................................... 9
   2.3 GUI frameworks ........................................... 16
   2.4 Multicast ................................................... 21

3 Implementation 25
   3.1 Development environment .................................. 25
   3.2 Management solution evaluation .......................... 33
   3.3 Implementation using SNMP ............................... 37
   3.4 Client GUI .................................................. 53

4 Results 62
   4.1 Development environment .................................. 62
   4.2 Management interface ...................................... 62
   4.3 Client GUI .................................................. 65

5 Conclusion and future work 66
   5.1 Future work ................................................ 66
   5.2 Conclusion ................................................ 66

References 68

A Glossary 70

B OMAP-MIB 72

C Mib2c example files 84
1 Introduction

The following introduction section is describing this project’s problem and puts it into context. The project’s goal criteria and limitations are discussed as well. Further there is an outline of the rest of this paper is given.

1.1 Context

In nowadays computers play an important role in many aspects of life. With the computer’s sizes and prizes getting lower and lower, while at the same time increasing processing power and memory, computers are used in ways, no one thought of just a few years ago. When a computer system, hardware and software, is designed for a special, dedicated task you speak of an embedded system. Often, embedded systems are not even recognized as such by their users, for example when you think of home stereo systems or wrist watches. This separates embedded systems from computers for desktop use, which are not built for a special purpose, in contrary these computers are built to perform a variety of tasks. This paper deals with the management of such an embedded system running the Linux operating system.

Bosch created a technique to stream live audio over a wireless network from one computer or embedded system to another. This technique creates the possibility to bring computers to another area of life. It can be used to provide an IP-based, digital solution for a public addressing systems for buildings. This could for example be the speakers spread throughout an airport used to make public announcements about flight schedule changes, security advices and the like. Almost all systems available today are using either analogue or non-IP-based digital techniques.

![Figure 1.1: Overview of the announcement system](image-url)
In Bosch’s system there are several management consoles, which are used by the staff to set up the speakers and to make announcements. So if someone would like to make an announcement, he would select the speakers where he wants to be heard and then speaks into a microphone connected to the console he’s working at. The digitalized announcement is then sent to embedded systems connected to the speakers throughout the building using a wireless network. Each speaker has one embedded system connected to it, which is used to receive the data from the wireless network. Aside from making announcements, the staff should also be able to change the settings of the embedded systems (e.g. volume, audio-sources, etc.), combine them in different groups, run system updates and so on.

1.2 Problem

Although the work on the technique used to wirelessly transmit live speech data to the embedded devices connected to the speakers is completed, there is no convenient way to manage the possibly high number of speaker devices in the network. Several problems need to be solved regarding different management aspects of these systems.

There is no out-of-the-box development environment available for an embedded system like this. Development environments for embedded systems usually need to be set up especially for the target system, because of the variety of different hardware used in these systems. Most embedded systems use special hardware and are more restricted regarding processing power and memory compared to standard desktop systems. Therefore it is required to create a development environment for the target system including a cross-compiler toolchain for the target architecture and supporting the selection of software packages suitable for an embedded system. The system should provide features to automatically create kernel and filesystem images for the target system.

Not only the obvious management tasks related to the live streaming of audio, but also tasks related to system management arise. This includes setting up IP-Addresses, giving different users different rights to access and change different settings on the speakers, rebooting a system and installing updates. All these management tasks should be done in a secure manner over the network, which means they can be done remotely from one of the management consoles without the need for physical access to the speaker systems installed in a building. Management activities should consume as less bandwidth on the network as possible, because the main use of the network should be to make announcements.

1.3 Objectives

This section describes the goal criteria for the various parts of the project.

- Creation of a development environment
  The development environment should support a developer in all tasks not related to the actual programming of software for the embedded system. Starting with the creation of the cross-compiler toolchain for the target architecture. In this project the development host is running SuSE Linux on an Intel based system and the target system is an ARM architecture. Therefore a cross-compiler, running on Intel, but creating code for ARM is needed.
  The development environment should provide automated builds for all software packages required for this project. But it should also be extensible, meaning it
should support the addition of new software to the development system. Once added, these new packages should build just as automated as the original ones.

Once all packages are built, they have to be placed into a filesystem image. These images are usually using a different filesystem (e.g. JFFS2), than desktop computers. In addition to installing the software into the proper locations, there has to be more work done to the filesystem, like creating the device nodes for the hardware available on the target. The creation of filesystem images should be done by the development environment in an automated manner, too.

The last step required for having a working system on the target is the creation of a kernel image of the Linux kernel suitable for the target’s hardware. The development environment should also solve this problem for the developer. This means it has to apply required patches to the official Linux kernel sources, compile the kernel for the target architecture and create an image of the kernel for the used bootloader.

- **Implementation of a management interface for the embedded systems**
  At first, an evaluation of different available solutions for the management of networked devices is necessary. The goal is to evaluate the properties of each solution regarding its requirements in memory and processing power and bandwidth consumption. Another important aspect is the security mechanism provided by a solution.

  During this evaluation it is necessary to keep in mind and have an idea about what management tasks arise while using the speaker system. In other words: What features should the interface provide to be ready for real-life usage?

  After this evaluation a management interface has to be implemented, using the solution that fits best.

- **Creation of a GUI driven management console**
  At last a management application providing a GUI, to apply the various settings on the speaker systems has to be created. It should run at least on Linux systems, but support for more operating systems would be a plus.

1.4 **Constraints**

The work has been restricted regarding certain aspects. Time has been an restricting factor. The project was planned for seven months and a result should be a proof-of-concept and not a system for real-life use. There was also no testing or choices between different hardware platforms. The project was limited to the hardware described in section 2.1.3. Five of the described systems were available for testing. A more final system would need more testing, regarding the number of systems used in tests, but also regarding the time spent testing in general.

1.5 **Outline**

Section 2 provides technical background information about the technologies used in the project. After an introduction to embedded systems and Linux, different solutions for system management will be presented. The hardware platform used for the embedded
system is shown as well. A brief discussion of different GUI frameworks and Multicast in general is also provided.

In section 3, the actual project work is presented. The section is divided into four subsections, one for the development environment, one the client GUI, one for the evaluation and one for the implementation of the management interface. Section 4 presents test results and user opinions. The last section shows insights gained during the project, as well as future developments.
2 Theoretical background

The following section gives theoretical background information on the technologies and terms that have been used in the project.

2.1 Embedded systems

As stated in section 1, many devices we are using in everyday life, which are equipped with a microprocessor are embedded system. Because embedded systems are in so many different aspects of life, there is a large variety of different kinds of systems. That includes watches, DVD players, navigation systems, medical equipment and so on. This section gives you an idea how development for embedded systems differs from development for other computer systems and describes the embedded hardware used for this project [2, 34].

2.1.1 Development for embedded systems

There are several aspects how development for an embedded system varies from software development for desktop systems. Most of these differences exist because of the different hardware used for embedded systems. Usually embedded systems have less processing power and memory than their desktop counterparts. Embedded systems with 8-Bit processors and only a few kilobytes of RAM are not uncommon. Therefore a developer has to write efficient code and always needs to keep memory usage minimal. This means static memory (the amount of memory needed to store an application on disk), but also dynamic memory usage (the amount of RAM consumed by an application at runtime) [2].

Although every program should be free of errors and robust, this is even more important for software running on embedded systems. As it usually would not be a big problem for users of desktop software to download and install an update, this will not be a convenient task on most embedded devices. Sometimes it is not even possible at all.

Most likely the toolset used for development will differ from the tools for the development of desktop applications. The general setup is that you will develop and build your software on a desktop computer or workstation, called the host. After the application has been built you have to download it to the embedded system, called the target. As said before, usually the target will use a different hardware platform than the host. This means, that you will need a cross-compiler. A cross-compiler works just as other compilers, with the difference, that it runs on one platform and creates programs for a different platform. You could be developing on an Intel x86-based host and your compiler (running on x86 hardware), will create programs for ARM-based systems [2].

2.1.2 Linux on embedded systems

The term Linux is often used for different things. It might refer to the Linux kernel, a Linux distribution or a Linux system. Strictly speaking Linux is the kernel maintained by Linus Torvalds, available for download at http://www.kernel.org. This is the bare kernel, without any other software. The kernel controls the hardware and provides processes, sockets, files, hardware access and so on to the software running on the system [2, 34].

But Linux is often also used to refer a whole system. If someone says, that he is using Linux they usually do not use only the kernel, but the kernel with a set of programs.
Many of these programs are provided from the GNU project. Users can either select all these programs by themselves and thereby compose their own system. Or they can choose to use a subset of the applications put together by some vendor or group. These collections of, often pre-built, software packages are called Linux distributions [34].

Most people running Linux on a desktop computer are using one of the many different distributions available on the Internet or in stores. Distributions usually come with some sort of package managing system (PMS). These systems allow the user to select and install the applications, that are included in the distribution. With such a PMS users usually do not have to worry about dependencies between the different software packages. The PMS resolves the dependencies by itself and selects additional software for installation if a program selected by the user depends on that software. It also keeps track of the different versions of packages. Just because a certain version of a package satisfies the dependencies of an application, does not mean that an older or newer version of the same package works, too. When using a custom system the user has to pay attention to these issues himself. Most likely he will also have to build and install the software himself. There are no pre-built packages installed by a PMS as when using a distribution.

When running Linux on embedded hardware, developers often can not use a distribution. Although there are some distributions available for use on embedded systems, you will most likely have some special needs, so that you have to build and select at least some software components by yourself. The high specialization of some embedded systems creates the need for a specialized software solution running on these systems. In most cases you will use programs developed especially for embedded systems, paying attention to memory consumption and required processing power, that solve the same problems as their counterparts on desktop systems. To achieve savings in resource requirements, the embedded versions usually omit some of the features of the desktop version. Often the developer can decide what features to build into the application and which not. Hardware differences might also cause the need for special software and libraries. The GNU standard C library for example needs a hardware platform with a Memory Management Unit (MMU). Every modern desktop computer has a MMU installed, so this causes no real restrictions on desktop systems. But embedded hardware may have an MMU or not, which in the latter case makes the use of this C library impossible [34].

As there is no such thing as the embedded version of the Linux kernel, you will most likely have to make some adjustments to a standard kernel. This means, that you have to apply patches to the kernel’s source code, thereby adding for example additional drivers to support the hardware of your embedded system [34].

In general Linux has the requirement for a 32-Bit CPU to work. Although there are projects working on Linux support for 16-Bit CPUs, all the main development is done for 32-Bits. So if you need to use a 16-Bit processor you will find much less support and Linux might not be the ideal choice for your project [34].

Roughly speaking, you will need to create the following things to be able to run a Linux system:

• A **bootloader**, suitable for your hardware platform.

• The **Linux kernel**, with special patches applied for your hardware if necessary.

• A **filesystem image**, containing at least a minimal set of applications needed during system boot.
There are different bootloaders available, which support different hardware platforms. Yaghmour [34], chap. 9 provides a list of the available bootloaders and the hardware platforms each of them supports. Often bootloaders are available as already pre-compiled images for their supported platforms. Otherwise you can download the source and compile it for your platform yourself.

As said before, it is likely that you have to make changes to the standard kernel to get a working kernel for your target’s hardware. There are open-source projects, adding support for a certain hardware platform, by providing kernel patches or already patched complete kernel sources for download on their own websites. You can either download a complete kernel source, already modified for the target, from a project’s website or you have to download the standard kernel source and apply patches provided by a project to it. Having the sources for your target platform, you might have to apply additional patches to make certain hardware (e.g. soundcards, network interfaces, etc.), not supported by the kernel out-of-the-box, work. In any case you will need a cross-compilation toolchain for your target platform to compile your kernel sources. (The tools contained in the toolchain, will be discussed later in this section) [34].

All the applications and resources you need to run your system will get stored in the filesystem image. You have to compile the required software using the cross-compiler, you already used to build the kernel sources. If some of the software packages you choose to use do not support (cross-)compilation for your target, you will have to make adjustments to the files controlling the build process of the source files. In some cases you will even have to make changes to the source code in order to make the compilation succeed. As embedded systems usually do not use harddisks as in desktop computers, you will have to use another filesystem (e.g. JFFS2, CRAMFS, ...) depending on your needs and the kind of hardware you are using.

Once all of these components have been built, you have to download them to the solid state memory of your target. Figure 2.1 shows a typical layout of an embedded system’s solid state memory after all components have been downloaded [34]. Usually, at least the bootloader has to be downloaded using a serial RS-232 connection. Depending on the available hardware and bootloader capabilities you can use an Ethernet connection to download the other images.

A cross-compilation toolchain consists of the GNU C compiler (GCC), the sources of the used kernel version, the GNU binutils package and a C library. Not all combinations of versions of these parts work together. You might have to try different combinations or use a combination, that is known to be working. Depending on your needs and the used hardware you might have to use a certain C library implementation. A list of functional version combinations is provided in Yaghmour [34], chap. 4. I will not discuss the build-process in detail, but this is the general procedure you have to go through after

![Figure 2.1: Typical memory layout of an embedded Linux system](image)
downloading the package sources:

1. Create kernel headers from the kernel sources
2. Binutils setup
3. Bootstrap compiler setup
4. C library setup
5. Full compiler setup

After all packages have been built, you have to install them on your host system and configure every package, you want to build for the target, to use these tools to compile their sources.

2.1.3 Used hardware

The TI OMAP5912 Starter Kit (OSK) from Texas Instruments and Spectrum Digital has been used as development target for this project. Figure 2.2 shows a photograph and table 2.1 lists the technical facts of the board. Figure 2.3 provides a block diagram of the board.

![Figure 2.2: The OMAP5912 board (top-view, image ©by Texas Instruments)](image)

The OMAP5912 is a dual-core architecture consisting of an ARM926EJ-S core for general purpose calculations and a TMS320C55x core, which is a DSP (Digital Signal Processor). Both cores are operating at 192 MHz. The ARM core’s features include a MMU and the jazelle Execution Environment, which speeds up loading and execution of Java Micro Edition (Java ME) applications. The board is equipped with 32 Megabytes of DDR RAM and 32 Megabytes flash memory for persistent data storage [12].

For connectivity the board features a RS-232 serial port, a USB1.1 port and a 10 MBit/s Ethernet port. It is also equipped with an CompactFlash card reader and several special expansion ports, e.g. to connect a display adapter. For input and output of sound there is an AIC23 audio codec on-board. The board offers connectors for headphones, line-in and microphones [12].
### 2.2 Management software

There are different solutions available, that enable network administrators to manage and monitor the devices in their network from a central station. Usually these solutions provide rules how information about the managed resources is modeled, how to transfer this information over a network and how to change settings remotely. A mechanism to ensure that only authorized personnel is able to view and especially change settings on the managed devices is also provided by most solutions. This section presents different standards for the management of resources in an IP-based network.
2.2.1 JMX
The Java Management Extension (JMX), developed by Sun Microsystems, offers a framework to developers for the management of applications, implementations of a services, devices, users and so on. To manage resources through JMX, a developer has to instrument them, using Java objects called MBeans. Once instrumentation has been done for a resource, it is manageable through a JMX agent. Agents control resources and provide access to them for remote management. Agents usually run on the same machine as the resources they are controlling. Apart from communication services for the remote access of management stations, a JMX agent consists of the MBean server and services for the management of MBeans. The managed resources, or their corresponding MBeans, need to be registered with the MBeans server [20, 19].

![Figure 2.4: The JMX Architecture](image)

The managed information can be accessed in different ways by a remote management console. JMX technology offers different protocol adaptors, for management through existing protocols, like SNMP or proprietary protocols. Connectors are used on the manager-side as handler for the communication between the management console and the JMX agent. When connectors or protocol adaptors are used, the management application can connect to an agent transparently, independent from the used protocol [20, 19].

2.2.2 NETCONF
Another way to manage networked devices is defined by the Network Configuration Protocol (NETCONF). It has been developed by the Network Working Group of the Internet Engineering Task Force (IETF). NETCONF uses remote procedure calls (RPCs) to establish communication between management stations and managed devices. Through
this mechanism configuration data can be retrieved, manipulated and new data can be uploaded. The configuration data is specific to different applications. As figure 2.5 shows, the actual configuration data represents layer 4 of NETCONF’s layer model. Because structure and type of the configuration data is specific to the managed application it can not be discussed here in greater detail [8].

Figure 2.5: NETCONF layering concept (based on ASCII image in [8])

Layer 3 defines a set of base operations that NETCONF enabled devices need to support. It is also possible to extend this set of operations, by defining capabilities, which describe new operations and their contents. The actions, defined as operations of capabilities, are encoded as XML structures. In layer 2 there are <rpc> and <rpc-reply> structures defined. All data of a request and a response are serialized to XML structures. That means the actual function call and the data being sent (e.g. computation results, configuration data, etc.) get converted into entries in an XML structure, which is enclosed by a <rpc>, or in case of an response by a <rpc-reply> tag [8].

Responsible for a communication path between client and server is the transport protocol layer. The transport protocol used for NETCONF is exchangeable as long as it fulfills some requirements [8]:

- Connection-Oriented Operation: NETCONF connections are long-lived, persisting between protocol operations

- Authentication, Integrity and Confidentiality: A NETCONF peer relies on the transport protocol regarding these aspects of a connection. It assumes, that a connection is secured by the protocol used in the transport layer.

According to RFC 4741 [8], each implementation of the NETCONF protocol must support the SSH protocol for transport. Using NETCONF with SSH, the managed device listens for incoming SSH connections and the management console initiates a SSH session. After the client has been authenticated and the SSH session has been established, NETCONF will be invoked by the client as SSH subsystem, called netconf. Access to this subsystem is only granted, when the session is running on a certain TCP port (the default is port 830). This makes easy filtering of NETCONF traffic on firewalls possible. After
the session has been established, client and server exchange XML documents, listing their capabilities. After this, the client can start configuring the server through NETCONF, using a secure connection provided by the SSH session [31].

Another protocol, defined by the IETF in RFC 4744 [14], for NETCONF transport is the Blocks Extensible Exchange Protocol (BEEP). BEEP uses the Simple Authentication and Security Layer (SASL) for authentication and Transport Layer Security (TLS) for encryption of the communication. By default NETCONF over BEEP is using TCP port 831 for connections. After a user has been authenticated and a private session has been established, the protocol works as described above, using SSH [14].

The BEEP protocol can also be used to send Simple Object Access Protocol (SOAP) messages for NETCONF sessions. Additionally to BEEP, SOAP can also be used over the Hypertext Transport Protocol (HTTP), or rather its secure version HTTPS [10].

2.2.3 WBEM/CIM

The Desktop Management Task Force (DMTF) created a data model to describe all kinds of managed objects. Not only physical objects, like servers and routers, can be represented with that model, but also logical entities and services hosted on the physical objects. The DMTF’s data model is called the Common Information Model (CIM). To have a complete management solution there was also the need for a communication protocol and an encoding standard for the management information. The DMTF chose to use web technologies to solve these problems. The combination of these technologies and CIM is called Web-Based Enterprise Management (WBEM) [33].

CIM is defined in a language called Managed Object Format (MOF). Using MOF, managed objects can be modeled in an object-oriented way. Abstraction and classification, as well as inheritance are supported by MOF. That means, that it is possible to define common properties of high-level, maybe abstract, objects in the management domain. Actual objects can then inherit common properties by subclassing of higher-level objects. It is also possible to model relationships between objects in CIM. By using relationships, you can create component associations and dependencies. You could for example have

![Diagram](image-url)
the fact, that some network card is part of a switch or that a web service is dependent on a working Internet connection to function properly, modeled in your CIM. Through the inheritance of common methods and their entity-specific implementation, a uniform interface is provided for the invocation of methods. An example could be a Reset method, which always is activated the same way, independent from the kind of device and its vendor [6, 7].

CIM is used to describe management information and the relationships that exist between the different entities. Another part of WBEM is xmlCIM. It defines how CIM is represented in XML format. This is necessary to send CIM information over a network. The specification describing how xmlCIM messages are exchanged over a network using HTTP or HTTPS for a secure message exchange, is called CIM-XML. Because of the use of HTTP, the connection oriented TCP is the basis for WBEM communication [28].

A server application that implements WBEM is called a CIM Object Manager (CIMOM). A CIMOM has to understand CIM, be able to interpret xmlCIM messages and offer an HTTP(S) service to clients. Server and client exchange HTTP(S) messages with encapsulated xmlCIM messages inside as shown in figure 2.6. There can be different kinds of clients connecting through the CIMOM. Clients have to be capable of generating HTTP or HTTPS requests for sending xmlCIM messages to the CIMOM. Another requirement for clients is to be able to encode and decode the xmlCIM messages. There are no restrictions, regarding the programming language used to develop a client application. It is however a good idea to use a language with good XML parsing and network communication libraries. There are also open-source API libraries available for popular programming languages like Java [28].

2.2.4 SNMP

The Simple Network Management Protocol (SNMP) was created, because there was the need for a systematic way to monitor and manage a computer network. Since the initial definition in RFC 1157, published in 1990, SNMP became the de facto standard for network management. Because of the shortcomings of SNMP version 1 (SNMPv1), a second version was defined. SNMPv2, in its latest revision, is defined in the RFCs 1441 to 1452. Because of the security flaws in SNMPv1 and SNMPv2 the third and latest version of SNMP was created. SNMPv3 adds mainly security features to the existing standard. It is defined in RFCs 3410 to 3418 and RFC 2576. All RFCs are available from the IETF website at [http://www.ietf.org/rfc.html][17, 29].

The SNMP model of a managed network consists of four components:

1. Managed nodes, running an SNMP agent.
3. Management information
4. A management protocol

An SNMP agent is a piece of software running on all devices on a network, that support to be managed using SNMP. Today, most network devices have some kind of agent built-in. Agents maintain a local database of variables to provide information about their managed resources to management stations [17, 29].
Management stations are general-purpose computers running a special management software. These managers are often called Network Management Stations (NMSs). An NMS is using the SNMP protocol to communicate with the agents (figure 2.7), by polling and receiving TRAPs. Querying the value of a managed object, is called a poll in this context. When doing a poll the communication is initiated by the NMS, by sending a request to an agent. The agent retrieves the requested value from its database and sends a response, containing this value, to the NMS. A TRAP on the other hand is sent from an agent to an NMS, without a request by the NMS. Traps are used to report events immediately, without the NMS needing to explicitly poll for values connected with the event [17, 29].

Managed objects and their behavior are defined according to The Structure of Management Information (SMI). A database with definitions of managed objects, using SMI syntax, is called the Management Information Base (MIB). SMI is using a subset of Abstract Syntax Notation One (ASN.1 [13]) to specify an object’s datatype. ASN.1 defines how data is represented and transmitted over a network. It is machine independent which means, that you do not have to worry about platform specific properties like byte ordering. A MIB provides a textual name for managed resources as well as a description of the resource. All agents have to implement a MIB called MIB-II, defined in RFC 1213. MIB-II is the successor to MIB-I which is deprecated and not used anymore. It defines variables like system contact, system location, interface statistics, running processes and so on. It thereby provides management information for TCP/IP based devices. In addition to MIB-II devices can support other, vendor specific, MIBs. These MIBs can be specific to a certain device, e.g. a particular router, or a service running on a machine [17].

Every managed resource has an unique object ID (OID) assigned in SNMP. Managed objects are organized in a tree structure. An object’s OID is defined by its position in
this tree. A part of the SMI object tree is shown in figure 2.8 (some parts of the tree are left out intentionally). OIDs consists of numerical values separated by dots, representing an objects position in the tree. For example the object called private(4) in figure 2.8 has the OID 1.3.6.1.4. A mapping of numbers to names is also possible, to be more human readable (e.g. OID 1.3.6.1.4 refers to iso.org.dod.internet.private). The tree structure is managed by the Internet Assigned Numbers Authority (IANA) and everybody can request branches in the tree for their own MIBs [17].

As figure 2.7 shows, SNMP is using UDP to exchange messages. This means, that messages are sent without previously establishing a connection and without the sender knowing if the message has actually been received [17].

In SNMP versions 1 and 2 community strings are used to manage who has access to an agent using SNMP. There are usually two community strings, the public and the private one. Using the public string grants you read-only access to a device’s managed objects; the private one grants read-write access. This means, that community strings are nothing else than passwords. In SNMPv1 and SNMPv2 community strings are sent in clear text over the network. This is a big security issue, since it is no problem at all for someone connected to the network, to capture the messages exchanged between a NMS and agents. Because the community strings are clear text, the attacker can read the community strings out of the captured messages, thereby gaining access to all SNMP agents that use this community string. Because of these flaws regarding security, SNMP version 3 was created. Aside from the security features, no new functionality is introduced by SNMPv3 [17].

In SNMPv3 managers and agents are called SNMP entities, consisting of an SNMP engine and one or more SNMP applications. Each SNMP engine consists of Dispatcher, Message Processing Subsystem, Security Subsystem and Access Control Subsystem. The
Dispatcher checks the version (SNMPv1, v2 or v3) of incoming messages and if the version is supported, it passes the message to the Message Processing Subsystem. Apart from checking received messages, the Dispatcher also sends outgoing SNMP messages [17].

The Message Processing Subsystem consists of multiple modules, one for every supported protocol. This means there are usually three modules, one for SNMPv1, one for SNMPv2 and one for SNMPv3 messages. The Message Processing Subsystem is responsible for preparing outgoing messages and extracting data from incoming ones [17].

The Security Subsystem is responsible for authentication and privacy. Apart from community strings, used in SNMPv1 and SNMPv2, SNMPv3’s user-based authentication is supported. With user-based authentication it is possible to create several users on an SNMP entity, in contrast to the public and private community strings supported in earlier versions. Also passwords are not sent as clear text, but MD5 and SHA algorithms are used to protect passwords when sending messages over a network. Using the Security Subsystem’s privacy service it is possible to encrypt the payload of SNMP messages as well. The DES algorithm is used for encryption by the privacy service [17].

Once a user has successfully been authenticated, the Access Control Subsystem is used to control which MIB objects the user is granted access to. It is also possible to define on a per OID or subtree level, if a user has read-only or read-write access to certain resources or no access at all [17].

2.3 GUI frameworks

Graphical User Interfaces (GUIs) are supposed to make it easier for users with little knowledge and experience of computers, to use computer software. The possibility to control a computer, by using a pointer device (usually a mouse) feels more natural to most users, than memorizing and entering a set of commands. A GUI usually consists of windows, icons, menus and a set of control elements. These control elements, like for example buttons or text boxes, are often called widgets.

GUI toolkits provide a set of basic building units a programmer can use to create a GUI-driven application. A programmer can use widgets provided by a toolkit to build a GUI. Most toolkits also provide methods to handle events, like telling the program what function to execute when a user clicks on a particular button. Usually there are also methods included, to support a programmer with making different localized versions (versions in different languages, for different countries) of an application. This section presents three popular GUI toolkits.

2.3.1 JFC / Swing

The programming language Java, created to support platform independent software development by Sun Microsystems, needed a toolkit for GUI creation. But the GUI framework of a language supporting that many platforms, can only support features provided on all platforms. Otherwise the platform independence will be broken. Sun Microsystems created the Abstract Window Toolkit (AWT) following the concept of the lowest-common-denominator among the feature sets of Microsoft Windows, Apple Mac OS and Motif on Unix systems. In AWT every component in Java is mapped to a component of the host platform. That gives applications the same look as other applications on that platform. The problem is, that there are not many common features supported by all platforms.
As a result, a big effort was needed by application programmers to create professional GUIs with AWT [30].

Knowing that AWT’s capabilities were limited, Sun’s engineers started to work on a new GUI framework. The result were the *Java Foundation Classes* (JFC), which are included in the *Java Standard Edition* (Java SE) since version 1.2. The JFC consist of the following parts (see fig. 2.9):

- **Swing-GUI-Components**: Many new widgets, which are, in contrast to AWT’s widgets, completely implemented in Java.

- **Plug-able look & feel**: JFC-based applications have the ability to change their components appearance at runtime, without having to restart the application.

- **Accessibility**: This API provides new ways of interaction with an application, especially for physically challenged people (e.g. magnification, speech-recognition, etc.).

- **Java 2D-API**: This library is using object descriptions and paths to combine complex objects and to draw them on the screen.

- **Drag & Drop**: Drag & Drop enables JFC-based applications to easily exchange data with other application, even with applications that are not Java-based.

The Swing components are an important part of the JFC and therefore the two terms are often used interchangeably. Compared to AWT, Swing provides a lot more widgets. Symbols can be added to buttons and labels and components can be transparent and of arbitrary shape, which was not possible in AWT as well [30].

![Figure 2.9: The JFC architecture](image)

But Swing is still based on AWT. AWT needs so called *peer-components* for the mapping of AWT components to native components on a target platform, mentioned above.
This means, that for example an AWT button has a peer from the host’s native GUI system, which is responsible for rendering the button on the screen. Although this technique limits the amount of available widgets, it performs quite well regarding memory consumption and execution speed [30].

Swing’s components are lightweight. That means that Swing’s components do not have peers. All components are painted on the screen using primitives, like squares and circles. This makes painting of arbitrary shapes, independent from the underlying operating system possible. Swing is still using an AWT peer for certain basic elements, to be able to draw a window’s frame. Inside of an application window the rendering is done by Swing. This approach is better than AWT regarding platform independence and more feature rich, but due to the non-native painting it is also slower and consumes more memory [30].

Developers using Swing to build a GUI for their applications can either design their user interface by writing Java code or they can use a development environment, which has a GUI designer. With a GUI designer, you can build a user interface just like in a paint program, by dragging user controls, like for example buttons, into the desired positions. After you have ’drawn’ your application’s GUI, you can save it and the designer will generate the Java code for you. A development environment, providing an interface designer supporting Swing, is NetBeans. It is available as free download at http://www.netbeans.org.

2.3.2 GTK+

Another GUI toolkit with support for multiple platforms is The Gimp Toolkit (GTK+). GTK+ is an open source project under the GNU LGPL. It has originally been developed as a widget set for the GNU Image Manipulation Program (GIMP). Since then the project has grown and has been used in many other open source applications. One of these applications is the well-known GNOME desktop environment (http://www.gnome.org).

It is based on several other open source projects (see figure 2.10) [4]:

- **GLib**: Cross platform utility library used by many open source projects. Until GTK+ version 1.2, GLib has been part of GTK+, but since then it is a separate package. GLib provides among other things, byte order handling functions and interfaces for event loops, threads, dynamic loading and the GObject system.

- **Accessibility Toolkit (ATK)**: A set of interfaces, to support disabled people in using GTK+ applications.

- **Cairo**: A vector based graphics library, supporting multiple platforms, like Windows, X11 and, although experimental, Mac OS X.

- **Pango**: Text layout and rendering library. Pango is used for text and font handling in GTK+ and provides support for localization of applications in different languages. Pango uses Cairo for text rendering, but depending on the underlying platform it supports other rendering interfaces, too.

In addition to the software packages shown above, there are several other packages required for building and running GTK+ and the libraries shown above. Those dependencies include pkg-config, FreeType, fontconfig, JPEG, PNG and TIFF image libraries [9].
The GTK+ layer, shown in figure 2.10, is the widget toolkit, providing high-level functions for creating windows, menu bars and widgets, like buttons and check-boxes, etc. GTK+ is using the GTK+ Drawing Kit (GDK) to draw widgets on the screen and to handle mouse cursors, events and Drag & Drop functionality. GDK is an intermediate layer, providing a wrapper for a platform’s windowing system (e.g. X11, Quartz,...) [4, 9].

Initially GTK+ has been developed to run in conjunction with Xlib, on X11-based Unix and Linux systems. Over time, support has been added for running GTK+ on Windows systems by Tor Lillqvist. The Windows port currently supports Windows 98/ME/NT4.0 up until version 2.6. The latest version (2.11) supports Windows 2000/XP/Vista. A company named imendio started a project with the goal to port GTK+ to Mac OS X. But as they state on their homepage, the ”port is not yet finished or usable for mainstream use” [9, 11].

Although GTK+ is written entirely in C, it has bindings, offering an object-oriented API, to lots of programming languages, like C++, Ada, Perl and Python, to name a few. This is possible due to GLib’s GObject system. It provides a generic type system, a collection of primitive data type implementations, a fundamental object implementation to base object hierarchies upon and a flexible signal system, which can be used for notifications [1].

With Glade, there is an interface designer for GTK+ available as well. Glade is an open source project, too. There are two ways to use Glade for the creation of GTK+ GUIs. In both cases you create a GUI by dragging user controls into your application’s dialogue windows in Glade. After you have created your dialogues, Glade can either generate source code for your dialogues or it can save them in an XML format. The source code will be compiled when you compile your application. When you are using XML files, the GUI will be created at runtime by libglade based on the information in
the XML files. It is recommended to use the combination of libglade and XML files, especially when working on large projects [9].

2.3.3 Qt

Qt is a GUI framework for cross-platform developed by the Norwegian company Trolltech [http://www.trolltech.com]. Qt uses a dual-licensing model featuring an open-source license (GNU GPL) for the development of open-source projects under the GPL and a commercial license for the development of proprietary software. Because Qt’s source is available, developers have the ability to understand exactly what Qt is doing and how it’s doing it. There is also a big Qt developer community, working on various Qt-based open-source projects. On the other hand Trolltech offers commercial support for paying customers using the commercial license. Popular examples for Qt applications are the K Desktop Environment (KDE) on the open-source side and Photoshop Elements by Adobe on the commercial side [3].

![Qt architecture and supported platforms](image)

Figure 2.11: The Qt architecture and supported platforms

Qt offers a cross-platform C++ API not only for the creation of GUI elements and event handling. It also offers a large library, including classes for networking, XML-handling, databases, OpenGL, multithreading and others. Qt offers a uniform API for all supported platforms. As figure 2.11 shows, Qt is interfacing with the platform-specific APIs on a lower level, so there is no need for platform specific adjustments from developers of Qt-based applications. Qt is supporting Windows up to Vista, Mac OS X 10.3 and newer and X11-based Linux and Unix systems. A developer writing an application, using the Qt API can compile his project for each supported platform, without any adjustments to the code. The application will be using the look and feel of the target platform, thereby integrating nicely with applications developed, using the target platform’s native API [3].

Qt’s API supports Drag & Drop and Accessibility features, to help disabled users. Localization is supported by the API, but Qt also includes a tool called Qt Linguist for a convenient translation of applications. Although you can develop GUIs by writing C++ code, you can also use Qt Designer to graphically create GUIs and forms. Another tool,
greatly supporting cross-platform development is *qmake*. Qmake is a cross-platform build-tool, which can be used to create projects for Visual Studio for Windows development, Xcode for Mac OS X development and GNU make on Linux and other Unix systems. This means, that developers can not only easily build their applications on all supported platform, but they can also use the standard development tools on the target systems. Qt Designer and Qt Linguist are written using Qt and therefore run natively on all supported platforms.

### 2.4 Multicast

Generally speaking, *multicast* gives an application the possibility to reach multiple recipients by sending a single UDP packet to a special IP address. The network copies and delivers the packet to all recipients belonging to the logical group of computers assigned to the IP address.

The figures 2.12 and 2.13 show a computer network. The squares represent computers and the circles represent routers. In both figures, the computer represented by a green square wants to send a message to all computers represented by red squares. The thin lines between circles and squares are network connections, while the thicker lines represent actual connections between the computers on the network. They mark the way messages are sent from sender (green) to receivers (red).

Figure 2.12 shows how the sender has to send one copy of the message for each receiver. Sending a message to only one receiver is called *unicast*. In this example, the sender has to send six messages, one for every receiving computer. To use unicast delivery, the sender has to address his message to a unicast address. Unicast IP addresses have to be unique on a network, so each address identifies one host.
Using multicast, the sender has to send only one message, which is not addressed to a unicast address, but to a multicast address or *multicast group*. As you can see in figure 2.13 the message is copied by the network nodes for every connected branch with hosts in the multicast group. For example at the first router the sender’s message is passing through, it gets duplicated, because this router has receivers on two network interfaces. This means that in figure 2.13 the green host is, instead of sending six separate, but identical, messages (unicast, fig. 2.12), sending one multicast message, reaching all six red hosts. Therefore, when it comes to sending the same data to a large number of receivers, multicast scales a lot better than unicast [16].

The routers, with hosts connected that want to receive multicast traffic, need to know about the hosts’ memberships in multicast groups. The router closest to a receiver is called a *leaf router*. Hosts inform the router about joining a multicast group using the *Internet Group Management Protocol* (IGMP), defined by the IETF in RFC 2236. If a router has received information of one of its hosts, that it wants to receive multicast traffic for a certain group, it sends join messages to adjacent routers, thereby grafting itself into the multicast tree. The tree’s root is the host sending data to a group. The tree is expanding from the sender, over routers (branches), to the leaf routers and finally to the hosts (leafs), that are members of a multicast group. By sending IGMP join messages, new branches and leafs are grafted into the tree. When all hosts connected to a leaf router have left a multicast group, this branch will be pruned from the tree again. How a multicast tree is built up is a complex topic and beyond this paper. However, Tanenbaum [29], section 7.7.5 and Makofske and Almeroth [16]. Appendix A discuss multicast routing and related protocols [16].

Multicast works only with UDP transport. It does not work with the TCP proto-
<table>
<thead>
<tr>
<th>Address Range</th>
<th>CIDR Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>224.0.0.0-224.0.0.255</td>
<td>224.0.0.0/24</td>
<td>Local Network Control Block</td>
</tr>
<tr>
<td>224.0.1.0-224.0.1.255</td>
<td>224.0.1.0/24</td>
<td>Internetwork Control Block</td>
</tr>
<tr>
<td>224.0.2.0-224.0.255.0</td>
<td></td>
<td>AD-HOC Block</td>
</tr>
<tr>
<td>224.1.0.0-224.1.255.255</td>
<td>224.1.0.0/16</td>
<td>ST multicast Groups</td>
</tr>
<tr>
<td>224.2.0.0-224.2.255.255</td>
<td>224.2.0.0/16</td>
<td>SDP/SAP Block</td>
</tr>
<tr>
<td>224.3.0.0-231.255.255.255</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>232.0.0.0-232.255.255.255</td>
<td>232.0.0.0/8</td>
<td>Source Specific multicast Block</td>
</tr>
<tr>
<td>233.0.0.0-233.255.255.255</td>
<td>233.0.0.0/8</td>
<td>GLOP Block</td>
</tr>
<tr>
<td>234.0.0.0-238.255.255.255</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>239.0.0.0-239.255.255.255</td>
<td>239.0.0.0/8</td>
<td>Administratively Scoped Block</td>
</tr>
</tbody>
</table>

Table 2.2: The multicast address space categories

col. In contrast to the connection-oriented TCP, the connectionless UDP does not offer congestion control, in-order delivery or reliability. Therefore it is mostly used for applications where packet loss is acceptable like the streaming of audio and video. TCP repairs lost packets by stopping the transmission of new packets and starting to retransmit lost packets until the reception all packets have been acknowledged by the receiver. After that, TCP starts to transmit new packets again. This is usually not a desired behavior for applications where delay is critical and not reliability, like streaming. Considering the potentially large number of receiving hosts in a multicast group, using this reliability mechanism (collecting acknowledgment for every packet from every receiver) is impossible. And how should the sending machine deal with retransmissions of lost packets [16]? 

<table>
<thead>
<tr>
<th>Address</th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>224.0.0.1</td>
<td>All systems on a LAN</td>
</tr>
<tr>
<td>224.0.0.2</td>
<td>All routers on a LAN</td>
</tr>
<tr>
<td>224.0.0.5</td>
<td>All OSPF routers on a LAN</td>
</tr>
<tr>
<td>224.0.0.6</td>
<td>All designated OSPF routers on a LAN</td>
</tr>
</tbody>
</table>

Table 2.3: Permanent Multicast groups on a LAN

Table [2.4] lists the IP ranges, that have been assigned for multicast use in IPv4 networks by the Internet Assigned Numbers Authority (IANA). The shown IP addresses are called class D addresses. Each address identifies a group of hosts. There are two kinds of addresses: permanent and temporary ones. A permanent address does not need to be set up explicitly. Table [2.4] shows a list of the permanent groups on a LAN. For example, every host capable of sending and receiving multicast traffic is automatically in the group with the address 224.0.0.1. Groups using temporary addresses need to be set up first. This is done by a host sending an IGMP ADD_MEMBERSHIP message including the temporary address it wants to use. If a host wants to leave a group it sends a DROP_MEMBERSHIP message [29].
To scope a multicast group the Time-To-Live (TTL) parameter of the IP header is used. Possible values for this parameter are integers between 0 and 255. When used for unicast packets the parameter is used to define how many routers a packet may pass before it gets discarded if it has not reached its destination yet. When used for multicast packets the TTL field limits the destinations a packet is delivered to. It was common practice when multicast was deployed to use the values 1, 16, 63 and 127 representing the local subnet, organization, regional and global scope, respectively. So using for example a TTL of 16 would limit the multicast group to hosts inside of an organization’s network. Routers at the borders of the network need to enforce this policy, by not forwarding traffic to an outside network. Other values might be used as well, as long as routers enforce the proper policy [16].
3 Implementation

This section describes the actual project work in four subsections. Each subsection presenting the work on one area of the thesis. Namely the development environment, the client GUI, the evaluation of management solutions and the implementation of the management interface.

3.1 Development environment

As said in section 1.3 the development environment should support the programmer in all aspects of development for the embedded system which is not writing the code of the actual application. After searches on the Internet, I’ve found out that there are commercial solutions like MontaVista Linux (http://www.mvista.com), but also free and open-source solutions like Buildroot (http://www.buildroot.org) which provide most of the desired features. Another open-source solution is OpenEmbedded (http://www.openembedded.org), which is using an own build system called BitBake. I have decided to use Buildroot, because it is inexpensive and easy to add new functions and software packages to the existing Buildroot system. I have chosen Buildroot over OpenEmbedded, because it is using GNU Make Makefiles as build system, which I’m already familiar with.

3.1.1 Buildroot

Buildroot is using a set of Makefiles to provide an easy way for developers to create a cross-compiler toolchain and filesystem images for an embedded target system. It supports a variety of target architectures like ARM, PowerPC, MIPS and so on. A complete list of supported architectures is included in the Buildroot distribution. Buildroot does not have any versioning, so I’ve downloaded the CVS snapshot from March 10, 2007.

After uncompressing the Buildroot archive, you can start to configure your toolchain and filesystem for your target. Buildroot offers the user a GUI driven interface similar to the interface used to configure the Linux kernel (see figure 3.1). Using this menu you have to set up your target’s architecture and choose between different versions of GCC, binutils and C library. You can also select which software packages you want to install on your embedded target.

To meet the requirements of embedded systems concerning main memory and disk space, Buildroot is using uClibc as C library. UClibc is an alternative to the GNU C library glibc. It provides most of glibc’s functionality, but omits seldom used functions. Because uClibc is compatible to the standards C89, C99 and SUSv3 most applications, that will compile against glibc will also compile against uClibc [34].

Creating a toolchain and root filesystem boils down to these commands:

```
make menuconfig
make
```

The first command will bring up the menu shown in figure 3.1. After you have set up Buildroot for your target using this menu, the second command will start the creation of the toolchain, all selected software packages and the root filesystem. Buildroot compiles software packages from source code. If the source code of some packages is not available in your Buildroot installation it will download the packages from the Internet for you automatically.

25
Aside of the directory layout shown in table 3.1, you can choose in the Buildroot menu what kind of filesystem images you want to build and where they should be stored. It is very convenient to let Buildroot copy your filesystem images for example to your TFTP directory after creation.

The following list gives an overview of the actions performed by Buildroot after the make command is issued:

1. Check if all applications required for the toolchain creation are available on the host (e.g. GCC, Bison). If not, prompt the user to install them and abort.

2. Check if selected source packages for the toolchain creation are in the dl folder. If not, download them.

3. Build the toolchain.

4. For each selected software package:
   - Check if the source is available in the dl folder. If not, download it.
   - Uncompress the source.
   - Compile the source, using the cross-compilation toolchain created earlier.
   - Install the application to the root filesystem.

5. After all software packages are compiled and installed: Create a filesystem image from the root filesystem.

6. Copy the filesystem image to the location specified by the user.
### Table 3.1: Directory layout of a typical Buildroot installation

<table>
<thead>
<tr>
<th>Folder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>build_target</td>
<td>Build directory for all packages and the root filesystem. Where target is the architecture you are building for.</td>
</tr>
<tr>
<td>build_target/staging_dir</td>
<td>Install directory for the toolchain.</td>
</tr>
<tr>
<td>dl</td>
<td>Directory where all source packages are stored.</td>
</tr>
<tr>
<td>docs</td>
<td>Buildroot documentation.</td>
</tr>
<tr>
<td>package</td>
<td>Build instructions for all software packages.</td>
</tr>
<tr>
<td>target</td>
<td>Target specific build instructions and scripts for the creation of filesystem images.</td>
</tr>
<tr>
<td>toolchain</td>
<td>Build instructions for the toolchain.</td>
</tr>
<tr>
<td>toolchain_build_target</td>
<td>Build directory for the toolchain. Where target is the architecture you are building for.</td>
</tr>
</tbody>
</table>

#### 3.1.2 Additional software

Although Buildroot comes with build scripts for a large number of software packages, there was the need to add more packages to fit the needs of the project. For example a special version of the Network Time Protocol server (ntpd) created by Bosch had to be running on the embedded boards. The source code of Bosch’s ntpd (bntpd) was available in a gzip-compressed tar-archive. The first step for adding bntpd to Buildroot was to copy this archive to Buildroot’s dl directory where the source code archives reside.

The build scripts have to be placed into a new subdirectory in Buildroot’s package directory. For bntpd a directory bntpd has to be created and a file called Config.in and another one called bntpd.mk have to be placed into this directory. The Config.in file contains general information about a Buildroot package. Bntpd’s Config.in looks like this:

```ini
config BR2_PACKAGE_BNTPD
  bool "bntpd"
  default n
  help
    Bosch Network Time Daemon
    This version works with a NTPv4 server which sends
    out broadcast messages in periodic intervalls.
```

This means, that by default the bntpd package should not be built and installed and gives a short description, which can be displayed to the user in the Buildroot menu. It is also possible to define dependencies to other packages in this file.

The second file, bntpd.mk, is the build script for the bntpd package. It is a standard Makefile which handles decompression of the source archive to a subdirectory of Buildroot’s build_target directory. It will also configure, compile and install the bntpd to the root filesystem. It is possible to do anything with the source code of a package in this build script. You might for example apply patches to the source code prior to its compilation or copy some standard configuration file to the root filesystem for this
package. It is also possible to delete unneeded parts of a package like documentation or certain executables to save disk space on the root filesystem image.

To let user added packages show up in the package selection list in the Buildroot menu, the file Config.in in the package directory has to be edited. For every package in Buildroot there exists a line in Config.in. To add bntpd to the package list the following line has to be added:

```
source "package/bntpd/Config.in"
```

In this master Config.in in the package directory exists a reference to the Config.in in a package’s subdirectory for each package contained in Buildroot.

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alsa-lib</td>
<td>User space part of the ALSA sound drivers.</td>
</tr>
<tr>
<td>alsa-utils</td>
<td>Collection of sound applications using the ALSA drivers.</td>
</tr>
<tr>
<td>bntpd</td>
<td>Network time daemon, modified by Bosch.</td>
</tr>
<tr>
<td>id3tag</td>
<td>Dependancy of madplay.</td>
</tr>
<tr>
<td>libsamplerate</td>
<td>Dependancy of sox.</td>
</tr>
<tr>
<td>libsndfile</td>
<td>Dependancy of pcm6cast.</td>
</tr>
<tr>
<td>lmixer</td>
<td>Command line mixer control.</td>
</tr>
<tr>
<td>madplay</td>
<td>MP3 player optimized for CPUs without FPU.</td>
</tr>
<tr>
<td>pcm6cast</td>
<td>Audio streaming server and client modified by Bosch.</td>
</tr>
<tr>
<td>sox</td>
<td>Audio playback and conversion utility.</td>
</tr>
<tr>
<td>wpa_supplicant</td>
<td>Utility for access to WPA protected WiFi networks.</td>
</tr>
</tbody>
</table>

Table 3.2: Software added to the Buildroot system

Table 3.2 shows a list of all software packages, that have been added to the Buildroot system throughout the project. Most of them are dependancies of applications, that are needed for the project. But due to the fact that Buildroot has not reached a stable version yet, there have also been few errors, that needed to be corrected. The init scripts provided by Buildroot for the Universal Device Daemon (udev), used by recent Linux versions, had to be corrected. With the uncorrected scripts udev failed to start. A similar problem existed with the configuration files of the periodic command scheduler `cron`.

3.1.3 Linux kernel

By default Buildroot does not support kernel builds. To add this feature to Buildroot I’ve decided to add a new entry to the Buildroot menu which lets the users select wether a kernel should be built and where the resulting kernel image should be copied to. I’ve added support for the kernel versions 2.6.18, 2.6.19 and 2.6.20. The user can use the menu to choose what kernel version to use and there is also a menu option to select a configuration file for the kernel. Using this configuration file, a user can store a kernel configuration to file and distribute it easily to other computers. A kernel configuration contains information about what architecture to use, what hardware to support in the kernel and what drivers should be compiled into the kernel or should be compiled as modules.
To create the kernel related menu options I’ve created a new Config.in file in the also newly created subdirectory kernel in the package directory. I’ve also added a reference to this Config.in file in the master Config.in in the package directory as described in the previous section. As with other software packages I’ve downloaded the kernel sources for each of the three supported versions from http://www.kernel.org and placed them into the dl folder. I’ve also created a build script kernel.mk to uncompress and build the kernel.

At this point Buildroot will build the kernel, copy the resulting image file to the location defined by the user and install modules to the correct location in the root filesystem. But due to the uncommon hardware used on the target board (see 2.1.3), kernels built from the standard kernel sources would not support much of it. To add support for this special hardware there have to be patches applied to the standard kernel sources. Patches for the OSK5912’s hardware are available on the Internet from http://www.muru.com/linux/omap. I’ve downloaded the patches for the supported kernel versions and placed them into the same directory as the kernel.mk script. After a modification of the kernel.mk script a patch will be applied to the standard kernel sources prior to their compilation.

Although the resulting kernel images support the hardware of the OSK5912 board, they are still not ready for use on the target yet. Because the board uses the UBoot bootloader for ARM devices, available as source from DENX Software Engineering (http://www.denx.de/wiki/UBoot), kernel images have to be modified to work with the bootloader. I’ve added a final build step for the kernel images to kernel.mk to perform these modifications with a tool contained in the UBoot package.

The following list sums up the steps performed by kernel.mk and Buildroot to create kernel images:

1. Check if selected kernel sources are available in the dl directory. If not, download them from kernel.org.
2. Uncompress sources of the selected kernel to build_target.
3. Apply appropriate patch(es) for the selected kernel version.
4. Build the kernel image and modules.
5. Install modules to the root filesystem.
6. Modify kernel image for usage with UBoot.
7. Copy kernel image to the location specified by the user.

3.1.4 Wireless LAN

The OSK5912 board does not have wireless LAN support on-board. To use WiFi anyways, an adapter should be connected via the boards USB port. Bosch provided the WL-5480USB-50 by AirLive (see figure 3.2), an IEEE 802.11g USB WLAN device. It supports WPA protection and is equipped with a 5dBi Dipole Antenna mounted on a R-SMA antenna connector. According to the manufacturers website, there are drivers for Windows and Linux available [32].
As said in the previous section Buildroot supports automated builds for versions 2.6.18, 2.6.19 and 2.6.20. Version 2.6.18 introduced changes to USB core modules of the kernel. In previous versions there has been no checking whether a USB hub can provide connected USB devices with the amount of electric current they are requesting. In the case of the WL-5480USB-50 and the USB hub integrated in the OSK5912, this caused the kernel to refuse to use the WLAN adapter due to insufficient power. The adapter requests 500 mA, but the hub provides only 250 mA. The amount requested by a device is stored by the manufacturer of the device and is often chosen very carefully. This seems to be the case with the WL-5480USB-50 as well, because it has already proven to work flawlessly with the OSK5912 with older kernel versions. I’ve therefore modified the USB core modules of the kernel to print a warning but to accept all USB devices, ignoring current requirements. Because of further developments in versions 2.6.19 and 2.6.20 I’ve had to create a unique patch for every supported kernel version. I’ve then stored these patches in the Buildroot package directory together with the build scripts of the kernel package. The patches will be applied to all kernels build with Buildroot automatically.

The Linux driver available for download on AirLive’s website (http://www.airlive.com) does not work with newer versions of Linux. According to the documentation, accompanying the driver it has been tested with versions up to 2.4.22 of the Linux 2.4.x branch and up to version 2.6.7 of the more current 2.6.x branch. Users, downloading the driver package from AirLive have to build the driver themselves. Compiling the driver source code fails for kernel versions 2.6.18 and newer. Although I’ve managed to modify the driver’s source for compilation with kernel 2.6.18 it performs very unstable. More precisely wireless LAN support crashes after a - as it seems - random amount of time after joining a wireless network. Clearly this behavior is not acceptable.

AirLive’s adapter is using the ZydAS ZD1211b chipset. ZydAS has been providing Linux drivers for this chipset, but after ZydAS was acquired by Atheros, the download page has been removed from their website. But ZydAS has been providing their drivers as an open-source project under the GPL. Although the vendor support for Linux for this
chipset has been terminated, the community provided updates for newer kernel versions based on the sources released by ZyDAS. But these sources do not support kernel versions 2.6.18 and above, too. In the beginning of 2006 the developer community of the ZD1211b driver decided to halt the development of the driver based on ZyDAS’ sources and to start over with a rewrite instead. This new driver is called zd1211rw and supports the ZyDAS ZD1211 and ZD1211b chipsets, as well as the Atheros AR5007UG, which is the same chipset as the ZD1211b [15].

Since kernel version 2.6.18 the zd1211rw is included in the standard kernel, which means that it is not necessary to apply any patches to the kernel sources in order to have support for ZD1211 and ZD1211b based devices. Due to the fact that the WL-5480USB-50 is a USB device it is necessary to load firmware to the device when it is plugged into the USB port. Before that it will be unusable, although the correct drivers are loaded in the kernel. The firmware images for zd1211x-based devices is available for download from [http://zd1211.sourceforge.net](http://zd1211.sourceforge.net).

On Linux systems using kernel 2.6.13 and later the loading of device firmware is done by udev. The following list gives an overview of actions performed by a Linux system when the WL-5480USB-50 is plugged in:

1. The USB core module selects the appropriate profile and hands control to the corresponding driver module (zd1211rw).
2. The zd1211rw driver determines whether it supports the device and what firmware to load by the device’s USB ID.
3. An event is sent to udev to load a firmware image.
4. Udev determines according to its ruleset what script is responsible for firmware-loading and executes it.
5. The loader script sends the specified firmware to the USB device.
6. When the transfer is finished, the device reboots into the uploaded firmware and is ready for use.

As mentioned in section [3.1.2](#), the udev installation provided by Buildroot was faulty. After changes to udev’s init scripts and the firmware-loader script the WL-5480USB-50 adapter is working properly and stable with kernel versions 2.6.18, 2.6.19 and 2.6.20 on the target board. I’ve also added the zd1211 firmware images to Buildroot, they will get installed into the proper locations once the users selects the support for zd1211x-based devices in the Buildroot menu.

### 3.1.5 Audio playback and streaming

Another major change to drivers in the Linux kernel was introduced by the switch from the Open Sound System (OSS), used in 2.4.x kernels to the Advanced Linux Sound Architecture (ALSA) in the 2.6.x kernel series. That switch happened for various reasons and ALSA provides several advantages of OSS, that are beyond the scope of this report. However, details can be found on the website of the ALSA project [http://www.alsa.org](http://www.alsa.org).

The advantage of ALSA concerning this project is the ability to freely set sound buffer sizes, required for the audio streaming done by the pcm6cast package.
The OSK5912 has the AIC23 chip on-board for sound in- and output (see 2.1.3). Support for the AIC23 is not included in mainline Linux kernels. It is added when the patches for the OSK5912’s hardware from [http://www.muru.org](http://www.muru.org) are applied. But ALSA requires the alsa-lib package as well as some configuration files to be installed in addition to a working driver in the kernel for applications to be able to use ALSA for sound handling. Therefore I’ve added the alsa-lib package and the configuration files to the Buildroot system. At this point Buildroot’s ability for dependency handling is useful. This means, that as soon as the users selects a package depending on ALSA (e.g. pcm6cast) for installation, the alsa-lib package and configuration files get marked for installation automatically as well.

Although ALSA was working right away for pcm6cast it caused problems when used for sound output with other applications like those contained in the alsa-utils package. While no error message was displayed there was no sound output on the speakers. I’ve contacted a developer of the AIC23 ALSA driver, who could reproduce the problem, but could not provide a solution. There seems to be a problem with the calculation of buffer sizes. Bosch’s version of the pcm6cast streaming client calculates its buffer sizes by itself. Other applications base their buffer size on values from hardware registers of the sound card. Reading these registers produces incorrect results. Due to this there is no audio output. I’ve modified aplay an application for audio file playback of the alsa-utils package to use a static buffer size. After that modification there sound was output on the speakers, but it was distorted.

Because some applications do not support ALSA yet and still rely on OSS for audio output, ALSA drivers provide an OSS emulation. Using this emulation it is possible to use OSS-based applications with ALSA drivers. Because of the problems with the calculation of buffer sizes I’ve decided to use the OSS emulation for audio output, with the exception of pcm6client which is using ALSA directly. I’ve made this decision due to the time constraints of the project and the fact that for all applications I’m using except pcm6client there is no advantage in using ALSA directly.

Another problem occurred with the playback of MP3 files on the target board. Buildroot includes the application mpg123 for MP3 playback. Due to the lack of a FPU on the OSK5912 board (see 2.1.3) the decoding of MP3 files has been too slow and the sound output distorted. So there was the need for a MP3 decoder using fixed point integer arithmetic. The MPEG Audio Decoder (MAD) project of Underbit Technologies ([http://www.underbit.com/products/mad/](http://www.underbit.com/products/mad/)) provides this feature. It is distributed as a shared library under the GPL called libmad. There is also a player application, called madplay, available, which is using libmad. Because playback worked fine with madplay using AIC23’s OSS emulation I have added it to Buildroot, as well as its dependancy libid3tag. Libid3tag is used for the handling of ID3 tags in MP3 files.

### 3.1.6 Summary

Based on the software from the Buildroot open-source project an automated build system has been created. The system is capable of building the cross-compiler toolchain, patching and building of kernel images for the target’s hardware and provides a dependency-aware selection and build system for all needed software packages for the project. To achieve this several new software packages and build scripts were added to the Buildroot system as well as support for additional hardware has been added to the standard Linux kernel.
by applying patches and manual changes. After all software has been built, using the previously created cross-compiler toolchain, a filesystem image is created ready for upload to the target.

3.2 Management solution evaluation

The next sections show an evaluation of the different software solutions for remote management presented in section 2.2. Each solution is to be tested regarding its usability in an embedded Linux environment (size in ROM and RAM, CPU requirements) and support for multicast communication. In a scenario where an identical message is to be transferred to a possibly large (1000+) number of network stations it saves lots of network bandwidth when this message is sent using multicast and not unicast mode (see section 2.4). Therefore it is considered a great advantage over other solutions, if a management application supports multicasting of messages. An additional plus is if there is an open-source implementation or at least a trial version of the software available. An open-source solution makes it possible to fully understand how the application is working and to make, if necessary, adjustments to the application. This is an important advantage besides the financial aspect over commercial solutions.

3.2.1 JMX evaluation

Because JMX is based on Java technology (see 2.2.1) an evaluation of not only JMX but also Java is necessary. Talking about Java it is important to distinguish between the Java language and the Java platform. The Java language is a highly object-oriented programming language. The Java platform stands for a class library and the Java Runtime Environment (JRE), including the Java Virtual Machine (JVM). Java programs are compiled into a byte-code and not like e.g. C programs into machine code, which is specific to some architecture. A JVM is necessary for executing this byte-code by interpreting it [26].

As described in section 2.1.3 the OSK5912’s ARM9 core features Jazelle technology, making it possible to run some byte-code instructions natively, without the need for interpretation and translation. To use this technology it is necessary to use a special JVM licensed by ARM to paying customers. The JVM distributed by Sun Microsystems with the Java Micro Edition (JavaME) does not make use of Jazelle. The JavaME is distributed under the Sun Community Source License (SCSL), which is free for students and research use. In order to build a commercial product, using JavaME, a commercial use agreement has to be entered into with Sun.

Without the possibility to execute byte-code directly Java applications have a significant performance disadvantage compared to C or C++ applications, which run without interpretation. The interpreted code is 10 to 20 times slower. To address this issue vendors have developed Just-in-Time (JIT) compilers, that compile byte-code into native code for the target architecture on the fly every time instructions are loaded into memory. Although JIT compilation improves execution speed, the time needed for compilation may be too costly. Especially when the application’s code does not contain loops or runs only once. To address this issue Adaptive JIT compilers have been developed. An Adaptive JIT compiler monitors execution and only compiles code that runs frequently, otherwise the code is interpreted [26].
But JIT compilation creates another issue. The amount of RAM needed when a JIT or Adaptive JIT compiler is used is higher than when the byte-code is interpreted. That’s because the native instructions are placed in RAM after compilation and native code is larger than byte-code (4 to 8 times larger). Vendors are offering Ahead-of-Time (AOT) compilers to deal with this problem. These compilers avoid compilation at runtime and generate native code, which is statically linked against the JVM. This increases the size of an application in ROM [26].

<table>
<thead>
<tr>
<th>Feature</th>
<th>Speed</th>
<th>ROM</th>
<th>RAM</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>free, if non-commercial</td>
</tr>
<tr>
<td>Jazelle</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>not free</td>
</tr>
<tr>
<td>JIT</td>
<td>Med+</td>
<td>Med+</td>
<td>Med</td>
<td>not free</td>
</tr>
<tr>
<td>Adaptive JIT</td>
<td>Med+</td>
<td>Med+</td>
<td>Med</td>
<td>not free</td>
</tr>
<tr>
<td>AOT</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>not free</td>
</tr>
</tbody>
</table>

Table 3.3: Performance of Java applications compared

This always leads to a trade-off between speed, size in RAM and size in ROM. Table 3.3 gives an overview. It compares the properties of different ways to execute a Java application to each other, it does not relate to native code, like for example generated by a C compiler. Note that Jazelle performance depends on the functions that an application uses. If it is mainly using functions, that the ARM processor understands natively, the execution is much faster than if mainly functions that need to be interpreted are used. The RAM consumption of Jazelle and Interpreter applications shown in the table assume, that the application is statically linked [26].

Aside from the described issues, that are introduced by Java, JMX is is a very flexible framework. The method of using Connectors and Adapters as communication interfaces with the agent, a developer is free to choose a communication protocol. This means, that JMX supports UDP-based, as well as TCP-based communication. With supporting UDP-based protocols, the multicast communication can be used to reduce network load. In addition to the actual management application, written using JMX, there is the need to install the JavaME, which uses, according to Sun Microsystems, 2.5 MegaBytes of ROM disk space and 2 MegaBytes in RAM [18].

Overall JMX (or Java) does not really fit for use in an embedded environment or at least is causing several complexities, which can be solved, but only with the use of commercial software. As this project can not really profit form the advantages of Java, JMX is not a good management solution in this context.

3.2.2 Netconf evaluation

There are two open-source implementations of the Netconf protocol available: Applied Informatics’ C++ Portable Components (POCO) framework and YencaP, a university project from the Université Henri Poincaré in Nancy and LOIRA-INRIA in Lorraine. POCO (http://appinf.com/poco) is distributed under the Boost Software License, allowing free non-commercial and commercial use of the software. It is a C++ framework which, among other things, can be used to build Netconf applications. It has one property, which disqualifies it as a solution on an embedded target: its size. The POCO
library with SSL support for secure communication has a size of 27 MegaBytes. If all
symbols are stripped out of the library the size can be reduced to a minimum of 7.6
MegaBytes, which is still too much in an embedded environment, which makes testing of
RAM usage needless.

YencaP is distributed under the LGPL and faces a similar problem. YencaP is written
in Python, an object-oriented script language. Besides the performance issues, that might
arise from the interpreted Python code, this introduces a problem with size. The size
of a Python (version 2.4.4) installation may vary between 2-3 MegaBytes for a minimal
installation and up to 24 MegaBytes for a complete installation. There is no documenta-
tion available which parts of Python are needed for YencaP and which can be left
out. As YencaP also depends on other software packages, using Python, it is neces-
sary to keep track of their requirements as well when reducing the size of the Python
installation. This makes the task of reducing Python’s size very complicated and time
consuming. One of the packages YencaP is depending on is called 4Suite, available from
http://www.4suite.org. It is used for XML-handling by YencaP, depends on Python
and does not provide any information about what parts of Python are required to use
4Suite, too. However, as a minimal 4Suite installation requires 6.2 MegaBytes it is no
longer necessary to find out about what parts of Python are required as a minimum. Even
if the 2 MegaBytes, Python requires in a best-case scenario are assumed, and all other
dependencies of YencaP are left out, the 6.2 MegaBytes of 4Suite and the 1.6 MegaBytes
of YencaP itself add up to 9.8 MegaBytes, which is even more than POCO requires. The
large amount of required memory in ROM makes an investigation of RAM requirements
needless as in the case of POCO.

There is a Netconf implementation available from 6WINDGATE Software (http://www.6wind.com) called Extensible Management System (XMS), which is targeted at
embedded devices and completely written in C. However, XMS is a commercial product
and supports only TCP-based, connection-oriented communication. This prohibits mul-
ticast communication, which presents a disadvantage when a large number of networked
clients should be configured.

3.2.3 WBEM/CIM evaluation

Under the WBEM implementations, there are three open-source projects: OpenWBEM
(http://openwbem.sourceforge.net), OpenPegasus (http://www.openpegasus.org
and SBLIM (http://sblim.sourceforge.net). In contrast to OpenWBEM, which does
not provide a release targeting embedded systems, OpenPegasus and SBLIM do provide
Pegasus Lite and SFCB (Small Footprint CIM Broker) for embedded use.

According to the OpenPegasus website it should be possible to build Pegasus Lite
with a minimal size of 4 MegaBytes. I managed, following OpenPegasus’ instructions for
size-reduction, to make a minimal build with 13 MegaBytes. I think it is necessary to
remove some of the applications and libraries that get built. But there is no list available,
which files can be removed under certain conditions.

SBLIM’s SFCB performed much better. It compiled to 1.2 MegaBytes with SSL
support and without deployment support, which is close to the 1016 KiloBytes without
SSL support claimed by the project’s website. These numbers relate to an installation
without DMTF’s standard CIM repository, which would add about 3 MegaBytes in size.
But the installation of the repository is not necessary if you only want to manage resources
not provided by DMTF’s repository. Because this project does not aim at managing standard network hardware like a router or similar devices, a new repository has to be created and the standard repository does not need to be installed. The quite good properties regarding the size in ROM, SBLIM and WBEM in general, do only support messages encapsulated in HTTP requests. HTTP is a TCP-based protocol and therefore connection-oriented protocol. As mentioned earlier, this makes multicast communication impossible.

3.2.4 SNMP evaluation

The de-facto standard for network management is the Simple Network Management Protocol (SNMP). Therefore there are many implementations of SNMP available as commercial products and as open-source projects. A popular and proven open-source implementation is provided by the Net-SNMP project [http://www.net-snmp.org](http://www.net-snmp.org). The Net-SNMP suite is released under the BSD license, which makes it possible to use and even change the software without any license fees or the need to publish your changes.

Net-SNMP supports the SNMPv3 protocol, which provides network security. It supports authentication of communication partners as well as privacy by encrypting exchanged messages themselves.

Net-SNMP is written in C and supports both, UDP- and TCP-based communication. When UDP-based transfers are used, SNMP supports multicasting on transport level. This means, that multicast can be used to deliver SNMP messages to the target network stations, but as SNMP does not explicitly support multicast communication, it is possible that multicast communication is not supported at a higher level, for example application level. However because Net-SNMP is an open-source project it is possible to make the necessary adjustments to the Net-SNMP suite to add support for multicasting.

Net-SNMP has already been available in the Buildroot distribution for easy compilation and installation on the embedded target. It’s only dependency is the OpenSSL library for certain encryption modes. After some optimizations to the build-script provided by Buildroot, a Net-SNMP installation requires 208 KiloBytes in ROM and 681 KiloBytes in RAM. These numbers relate to a stripped, minimal agent without the standard MIB-II modules. All documentation and applications needed for development or on the client side have been removed. TCP-based transport has been disabled as well as all debugging support. This makes Net-SNMP perform very well in an embedded environment.

3.2.5 Conclusion

I have decided that Net-SNMP fits the needs of the project best. It is a free and open-source solution, providing secure communication and authentication when SNMPv3 is used. It is implemented in C and does not have a complex dependency structure. This ensures that performance and memory requirements are easily met by embedded systems. Although, there are necessary modifications to the SNMPv3 protocol, the UDP-based transport and therefore multicast communication present an advantage over all other solutions except JMX with its exchangeable protocol adaptors. The BSD license, Net-SNMP is using, makes it possible and easy to use and modify the application without any license fees.
3.3 Implementation using SNMP

This section discusses the creation of a management interface using Net-SNMP. First considerations about what settings and features of the embedded targets should be manageable remotely are presented. After these settings are explained, their implementation as a MIB module, as well as the generation of MIB modules in general, is described.

The section also presents how messages are exchanged between SNMP clients and agents, as well as what modifications and extensions were necessary to allow multicasting of SNMP messages.

3.3.1 Management Information Base syntax

As shown in section 2.2.4, SNMP’s management information is described in database called the Management Information Base (MIB). The syntax used, to define name, data-type and behavior of a managed objects is a subset of the Abstract Syntax Notation One (ASN.1). The information about a managed object also includes a short description of the object.

An example:

```plaintext
volume OBJECT-TYPE -- 1
SYNTAX Integer32 (0..100) -- 2
ACCESS read-write -- 3
STATUS mandatory -- 4
DESCRIPTION -- 5
"main playback volume" -- 6
DEFVAL { 76 } -- 7
::= { sys-settings 1 } -- 8
```

The above example defines an object named volume, which is described as the setting for the ”main playback volume” of some device. Note, that in ASN.1 all text after a double dash is interpreted as a comment. The comments in the above example represent line numbers for easy reference in the text. So the node’s name is defined in line 1 and its relative position in the OID tree (see section 2.2.4) is defined in line 8. Line 2 sets the objects data-type to integer and says that this object’s values can not be no smaller than 0 and no bigger than 100 to be valid. A default value for the object is also given (line 7). Line 3 grants read and write access to this object and line 4 says, that all SNMP agents need to implement this object.

But with the above example the position of the object volume in the tree and therefore its OID is not defined completely. Line 8 only defines volume as the first child of sys-settings. To define this object’s absolute position a few more lines are necessary:

```plaintext
IMPORTS enterprises FROM SNMPv2-SMI; -- 1

bosch OBJECT IDENTIFIER ::= { enterprises 5318 } -- 2

omap MODULE-IDENTITY -- 3
LAST-UPDATED "200708201200Z" -- 4
ORGANIZATION "Robert Bosch GmbH, Corporate Research" -- 5
CONTACT-INFO "external.andre.eisenmann@de.bosch.com" -- 6
```
Line 1 imports the enterprise node. Through this import the position or OID of enterprises is "known" (.1.3.6.1.4.1). The object bosch is defined as child number 5318 of enterprises (line 2) and has a child omap (lines 3 and 8). Additional information about omap is given in lines 4 to 7. In line 9 the relation between the node sys-settings from the previous example and omap is defined. This defines volume's OID as: .1.3.6.1.4.1.5318.1.1.1.

Every object is defined in the following format according SMI version 1 (RFC 1155):

```
<name> OBJECT-TYPE
SYNTAX <data type>
ACCESS <read-only, read-write, write-only or not-accessible>
STATUS <mandatory, optional or obsolete>
DESCRIPTION "Textual description of this object"
DEFVAL { <default value> }
 ::= { <Unique OID that defines this object> }
```

SMI version 2 extends this definition and is described in RFCs 2578 and 2579. Information about different data types and textual conventions is beyond this paper, but can be found in RFC 1155 and RFCs 2578 and 2579. The custom MIB module (see appendix B) defined in this project is using only the shown definitions.

Developers use the syntax described in the previous section to define custom MIB modules. To create a management interface for the embedded speaker systems using SNMP it is necessary to define the managed objects in a MIB file and to implement this module in a programming language like C.

### 3.3.2 Custom MIB module definition

As described in section [1.3] there have to be considerations about what features and aspects of the embedded system should be manageable remotely. I've grouped the managed objects into four groups, which I will describe in the following paragraphs, but can also be viewed in SMI syntax in appendix [B]

The first group, named sys-settings describes system related settings like executing commands and starting and stopping of services. The sys-settings group contains seven objects:

1. **volume**: Master output volume
2. **reload**: Issue a restart of the SNMP agent
3. **reboot**: Reboot the whole system
4. **exec**: Execute a command on the target
5. **mgroup**: Multicast group of the target
6. **delay**: Response delay used for multicast communication

7. **upload**: Start/Stop the SSH service on the target

All of these settings can be either written or read by a client application, by sending a SET or a GET request (see section 3.3.4). A GET request for the `volume` property returns the current volume on the target system, a SET request will set the volume on the target to the requested value.

A GET request for the `reload` property will return the number of restarts of the SNMP agent since the last reboot of the system. Setting this property to "1" will cause the agent to reload. Similarly, setting the `reboot` property to "1" end in a reboot of the system.

The string sent in a SET request for the `exec` property will be executed on the target system and the next GET request will contain the return value of this command. This is useful to perform tasks out of the ordinary on agents. Although, for more complex tasks it is more appropriate to use a SSH connection, this property can be very useful to perform simple tasks on multiple agents. Using a SSH connection it is necessary to log into every agent one-by-one to perform the task. This can be very time consuming if the number of agents is large. Using the `exec` property it is possible to perform the same command on multiple agents at once. Wether the command succeeded can be checked by viewing the return value of the command, using a GET request for the `exec` property.

Agents can be grouped into multicast groups using the `mgroup` property. If a SET request with a new multicast group is registered at an agent it tries to leave its old group and to join the new group. If joining the new group fails, the agent re-joins the old group. A GET request always returns the current multicast group an agent is in.

The `delay` property is used when agents receive multicast requests. It is described in more detail in section 3.3.5.

The SSH service can be started on a target agent by sending a SET request with "1" or stopped by sending a request for "0" for the `upload` property. A GET request always contains the current status of the SSH service on a target. SSH can be used to get a secure remote login to a target system, but it can also be used to exchange files with a target system using SCP. These files might be audio files uploaded to the target for playback or log files downloaded from the target.

The second group deals with the management of live audio streams. As the systems supports four simultaneous live streams at the moment there are four stream groups named `stream1` to `stream4` with the same contents describing the properties of a stream:

1. **streamName1**: Name or description for this stream
2. **ipAddr1**: IP address of the multicast group this stream is using
3. **portNum1**: Port this stream is using
4. **volumeMod1**: A modifier for this streams volume
5. **delay1**: Delay for this stream
6. **activate1**: Activate/Deactivate this stream’s playback
Note that the properties above have the suffix "1" because they are in the stream1 group. Respectively streamName2 would be the first property of the stream2 group and so on (see appendix 3). Therefore I’ll omit the numbers in property names in the following paragraphs.

A stream’s name or a short description can be set by sending a SET request for the streamName property of the corresponding stream. A GET request delivers the current value of this setting. This can be useful to keep an overview about what stream are currently set up on an agent. It is also possible to identify different streams by their IP address and port number, but using streamName makes this association more human-readable and does not require a user to know which stream is behind a certain IP address and port number.

However, for live streaming to work it is technically necessary to set up IP address and port of the stream. This can be done using the ipAddr and portNum properties. SET request can be used to assign new values, GET requests will return the current settings.

The playback volume for a certain stream can either be lowered by sending a SET with a negative integer value or raised by sending a positive value for a stream’s volumeMod setting. The delay property is used to keep the playback of a stream synchronous among the multiple speakers, that are playing the stream. GET requests for volumeMod and delay return their current values.

After a stream has been set up, setting the activate property to "1" will start the actual playback of the stream. Setting activate to "0" will stop its playback. If a stream is currently playing or not can be decided by sending a GET request for the activate property of the stream.

Note that it is up to the implementation what will happen to a stream, which is currently enabled using the activate property, if one of its settings is changed. This kind of dependencies are not defined in a MIB module’s definition.

The third group, playback, contains properties related to a scheduled playback of audio files. It consists of five items:

1. filename: Name and path of the file to play
2. pTime: Control/Schedule for single playback events
3. recurring: Add/Delete recurring events
4. listEvents: List scheduled events
5. fileCheck: Check files for scheduled events

The two properties filename and pTime can be used in conjunction to set a one-time playback event. A SET request with path and filename, e.g. /sound/alarm.mp3, for filename can be used to set a file for playback. To see which file, if any, is currently set for playback a GET request for filename can be issued. The property pTime is then used to set the time for the playback of the file defined by filename. To control playback or to define a playback time a SET request containing a string has to be sent to the target agent. If the string equals "1" playback will start right away, "0" will stop playback at once. It is also possible to schedule a playback event for a certain day and time. Therefore a string of the format "MMDDhhmm" has to be sent in the SET request, where MM
defines the month, DD the day, hh the hour and mm the minutes of the event. Sending
"0" will also cancel any previously set up events.

To schedule files for playback more often than once, but for example every day at
eight o’clock in the evening, the properties recurring and listEvents can be used. A SET
request containing a string of the format "[a/d]:mm:hh:MM:WeekDay:FileName" will
add or remove an event depending on whether the first character of the string is an "a"
for add or a "d" for delete. If "d" is requested, the agent tries to find an event, matching
exactly the rest of the string and removes it if found. If no such event is found it does
nothing. In the string mm stands for minutes, hh for hours, DD for the day of the month,
MM for the month, WeekDay is the day of the week and FileName is the name and path
of the file to play. All parameters regarding the playback time can also be set to "*",
meaning "every", but ranges like "5-7" may be given as well. The string "a:00:22::*:1-
5:/closingTime.mp3" would cause an agent to create ("a") an event that will play the
file "/closingTime.mp3" every day from Monday to Friday ("1-5") at 22:00 o’clock. This
will happen regardless of the month and day of the month, because both parameters are
set to "*". If the creation of this event succeeds, a GET request will be "0", if it fails it
will be set to "1".

To check which events are currently scheduled on an agent a combination of SET
and GET requests for listEvents can be used. Sending a string with an integer value
"x" in a SET request will cause the agent to deliver the event number "x" as re-
sponse to the next GET request. The answer will look very similar to the string used
for recurring. Only the leading "a" or "d" are missing, leaving a string with format
"mm:hh:DD:MM:WeekDay:FileName" as response. If "x" events are scheduled on an
agent and event number "x+1" is requested, the response will be "NONE".

It is also possible to check if all files used in scheduled events are available and acces-
sible. This test can be run by sending a GET request for fileCheck. As the MIB file in
appendix B shows, this is a read-only property. This means, that only GET requests are
allowed. SET requests will cause an error. The response to a GET request is either "0" or
the number of the event associated with the missing or inaccessible file. In combination
with listEvents this number can be translated to the file that is causing the error.

The last group, labeled aGuidance, is used to setup a feature called Acoustic Guidance.
It can be used to guide persons in a evacuation situation to the emergency exits of a
building. A person seeking an exit can “follow” the alarm signal to an exit. This works,
because the alarm signal seems to be moving towards the exit. Figure 3.3 shows how the
volume of the alarm signal develops over time. At the time t5 the whole sample of the
alarm, which starts at t0, has been played completely. While the sample is played it is
getting louder and lower. At t1 the volume is increased until t2, it is then staying on that
level until t3, where it starts to decrease until t4. If there are several speakers mounted
along a corridor, that start to play the alarm sample the later, the closer they are to
the exit, the signal will appear to move towards the exit. Using the properties in the
aGuidance group the values of the points marked as t0 to t5 can be set (SET) or retrieved
(GET). Acoustic guidance is not yet working and not part of this thesis. However the
settings required for acoustic guidance can already be managed remotely through SNMP.
But setting values of the acoustic guidance group does not have any effect other than
changing a variable inside the SNMP agent, no external applications are launched or set
up, as they do not exist yet.
3.3.3 Custom MIB module implementation

To add a custom MIB module to the Net-SNMP agent it is necessary to implement the module in C and to include it into the build of the agent at compile time. Net-SNMP is using the well-known triple `configure`, `make` and `make install` to create the binary files of agent and client applications from source. To have a custom MIB module built into the agent it is necessary to set the `--with-mib-module="<module-name>"` parameter when calling the configure script. This will change the Makefile responsible for control of the agent’s build steps, to look for files named `<module-name>.h` and `<module-name>.c` in the agent’s `mibgroup` directory.

Net-SNMP contains a code-generation application, named `mib2c`, to help developers with the implementation of custom MIB modules. It will parse the MIB file’s SMI syntax and create `<module-name>.h` and `<module-name>.c` containing skeleton functions for the managed objects in the MIB file. I’ve used mib2c to generate the code skeleton for the custom MIB module described in the previous section.

Using mib2c custom MIB modules can be created very fast, because a developer only has to implement the skeleton functions for GET and SET operations. The following paragraphs demonstrate by implementing the managed object `volume`, presented in the previous section, how mib2c works, how requests are handled in a module and what a module implementation has to look like.

The MIB module `omap` consists of the group `sys-settings` containing a single property `volume` stored in SMI syntax in the file OMAP-MIB.txt (see appendix C). To create the skeleton C files for the MIB file, the following command has to be entered to the command line:

```
MIBS=OMAP-MIB.txt mib2c -c mib2c.scalar.conf omap
```

The environment variable `MIBS` needs to be set to the custom MIB file `OMAP-MIB.txt`, that mib2c knows where to find the definition for the node `omap` (given as the last parameter), which it should create code for. The `-c` parameter is set to `mib2c.scalar.conf`, because the `omap` module contains the managed object `volume`, which is a scalar data type. This parameters needs to be set according to the kind of data you are handling.
in your module and where the data is stored. If the data is stored in the agent another setting is required than if the data is stored elsewhere, for example inside the kernel.

Mib2c has created the files omap.c and omap.h (see appendix C). The code contains a handler function for every managed object as well as an initialization function for the module. The initialization function is registering callbacks for the handler functions and is setting up OID values for the managed objects:

```c
void
init_omap(void)
{
  static oid volume_oid[] = { 1, 3, 6, 1, 4, 1, 5318, 1, 1, 1 };

  DEBUGMSGTL(("omap", "Initializing\n");

  netsnmp_register_scalar(netsnmp_create_handler_registration
    ("volume", handle_volume, volume_oid,
    OID_LENGTH(volume_oid), HANDLER_CAN_RWRITE));
}
```

The handler functions are always named `handle_<managed_object>`, so in this example, there is one handler function named `handle_volume`. This function is called whenever there is a request for the `volume` property. The handler functions contain a `switch()` statement to act according to the type of request (GET or SET). For example the code run for a GET request looks like this:

```c
switch (reqinfo->mode) {
  case MODE_GET:
    snmp_set_var_typed_value(requests->requestvb, ASN_INTEGER,
      (u_char *) /* XXX: a pointer to the scalar’s data */
      ,
      /*
       * XXX: the length of the data in bytes
       * /
    );
    break;
  case ...
}
```

Mib2c has already generated most of the required code and has placed placeholders where changes have to be made by the developer. Assuming, that the current volume is stored in an integer type variable, named `iVolume`, the fully implemented portion of code might look like this:

```c
switch (reqinfo->mode) {
  case MODE_GET:
    snmp_set_var_typed_value(requests->requestvb, ASN_INTEGER,
      (u_char *) /* XXX: a pointer to the scalar’s data */
      ,
      /*
       * XXX: the length of the data in bytes
       * /
    );
    break;
  case ...
}
```
This causes the value of iVolume to be encoded by the rules for the ASN.1 type INTEGER and to be added to the response message for the GET request.

Similarly the generated code for the SET requests for volume looks like this:

```c
case MODE_SET_ACTION:
    /*
     * XXX: perform the value change here
     */
    if ( /* XXX: error? */ ) {
        netsnmp_set_request_error(reqinfo, requests, /* some error */);
    }
    break;
```

Again the most work has already been done and the full implementation might look like this:

```c
case MODE_SET_ACTION:
    /*
     * XXX: perform the value change here
     */
iTmpVolume = *(requests->requestvb->val.integer);

    if ( iTmpVolume < 0 )
        iTmpVolume = 0; //clip volume at 0
    if ( iTmpVolume > 100 )
        iTmpVolume = 100; //clip volume at 100

    // set volume to requested value
    sprintf(mixerCommand,"lmixer -c pcm -v %i",iTmpVolume);
    sysRet = system(mixerCommand); // call lmixer to set new volume

    if ( sysRet != 0 ){
        // failure -> generate error
        netsnmp_set_request_error(reqinfo, requests, SNMP_ERR_GENERROR);
    } else {
        // success -> update variable
        iVolume = iTmpVolume;
    }
    break;
```

The requested value is temporarily stored into an integer variable, using the request structure, which is a handle to the received request. After some value checks the system’s
volume is updated using a shell command. The return value of this shell command is evaluated to decide whether updating the volume succeeded. A return value, stored in the variable \textit{sysRet}, other than "0" indicates an error. In case of an error message is sent as response, otherwise a success response is sent implicitly by the agent. If the shell command was successful, the variable \textit{iVolume} containing the current volume needs to be updated with the new value as well.

The implementation of all properties in the project’s MIB file (see appendix B) was similar. One thing out of the ordinary, I want to discuss in more detail is the implementation of the properties in the \textit{playback} group. Because the SNMP agent does not provide any schedule or alarm mechanism by itself a solution had to be found to ensure reliable event handling. One solution might have been to write a completely new application that provides scheduling. But as this would have been very time consuming and not really an objective of this project, I’ve decided to make use the \textit{cron} service which is available on most Linux systems.

An implementation of the cron service has already been included in Buildroot. Therefore installation and setup was fast and easy. I’ve configured the files controlling the system startup for the embedded target to start cron automatically. Once cron is running, it executes commands by a schedule defined in its configuration file. The syntax of the configuration file is quite similar, to the syntax used for the \textit{recurring} property, described in the previous section. An example of a cron configuration file used for the project, named \texttt{/etc/crontab} might look like this:

```
30 9 3 10 * /usr/bin/madplay /sound/speech.mp3
# below this are recurring events
00 22 * * 1-5 /usr/bin/madplay /sound/closingTime.mp3
```

The first line of the file always contains the event that can be set using \texttt{filename} and \textit{pTime}. There can be only one event of that kind at any time. A SET operation will overwrite an existing event, regardless if it has been executed yet or not. If there is no such event scheduled a the moment the line contains only the character "\#". That character tells cron to ignore the rest of this line and it can therefore be used for comments. The comment in the second line is the marker for the beginning of the events that are scheduled using the \textit{recurring} property. There can be any number of events of that kind.

The cron configuration above would schedule two events. The first line schedules the playback of the file \texttt{/sound/speech.mp3} on the 3rd of October at 9:30 in the morning. As cron does not distinguish between different years, this event would be executed every year. The second event causes the file \texttt{/sound/closingTime.mp3} to be played every workday at 10:00 in the evening.

Most handler functions in the \textit{playback} group parse \texttt{/etc/crontab} and add, remove or modify lines, thereby adding or deleting events. This way the well known and reliable cron service is used as a scheduling service for playback events of the custom MIB module.

After the C files generated out of the MIB file have been fully implemented and compiled into the agent, it is possible to send GET and SET requests to the agents, which will perform the requested actions and send a response back to the requesting client. A programmer does not need to know the structure of the actual messages exchanged by clients and agents to extend an agent with a custom MIB module. All the encoding of
values and the creation of request or response messages are done by the SNMP library libsnmp.

3.3.4 SNMP Communication

With the implementation of the MIB module described in the previous section it is possible to communicate with agents in unicast mode. But as described earlier, the multicasting of requests is more efficient than unicasting. This section will discuss SNMP unicast communication and the different request types in more detail. There will be only those message types discussed, that are relevant to the project. A discussion of all message types is beyond this paper. The next section will explain, based on the current section, the changes that I have made to the SNMP standard to make multicast communication possible.

Agents running SNMP versions 1 or 2 are using community strings for authentication of requests. There is a public and a private community, which are nothing else than two passwords. The public one grants read-only access and the private one read-write access. Not bad enough that you can not control to what settings a person has access on an agent, these passwords are also sent in plain text with every SNMP request. Because of these shortcomings SNMP version introduced the User-based Security Model (USM) and the View-based Access Control Model (VACM), as well as secure authentication by using encryption. Using USM an administrator can define several user groups and users on an agent. For these users and groups access can be granted to only certain settings. For example an administrator can grant the person responsible for management of the router functions of a server read-write access to these settings, but not to the settings for the web-server at all. With SNMPv1 or SNMPv2 it was only possible to grant either full read-only, full read-write or no access at all to all settings of an agent. Therefore all SNMPv3 messages contain values for user name and password.

Another big security flaw of SNMPv1 and SNMPv2 was the transmission of community strings or passwords in clear text. This means that depending on how often requests are made by an administrator it is getting more and more likely that someone, using a program that monitors network traffic, captures a SNMP request and therefore knows the community string used in that request. SNMPv3 provides two mechanism to protect requests against an attack with a program capturing packets of the network. The first method is providing a secure authentication by not sending the password in clear text. Net-SNMP provides the SHA-1 or MD5 algorithms for password protection. However, only the password is protected using this method. The actual data of the request, like which value is requested and in case of a response also the current value of the requested property are send in clear text and can therefore be read by someone capturing network traffic. If this information should be protected as well, Net-SNMP is providing encryption of the payload using the DES algorithm.

To provide protection against replay attacks SNMPv3 is using the properties engineID, engineTime and engineBoots, which are sent with every SNMPv3 message. Figure 3.4 shows an SNMPv3 message. If someone is capturing a packet from the network and is re-sending it, this is called a replay attack. You could for example capture a SNMPv3 SET message. If secure authentication is used, you will not know the password but the type of the message and the targeted property of the request. If privacy, this means payload encryption is used as well, you will only know that this is some SNMP request
Figure 3.4: Diagram of a SNMPv3 message

from the host with IP X to the host with IP Y. Although you do not know the password and therefore can not create your own requests out of the information you’ve captured, you could just re-send the request you’ve just captured. Assuming that the password used in the captured message was correct, an agent having the same user and password will accept the message and perform the requested operation. To counter these attacks SNMPv3 is using engineID, engineTime and engineBoots. EngineID is a hexadecimal value assigned to an agent or client which should be unique in an administrative domain. Every application capable of performing SNMP actions is called an engine in SNMP vocabulary and has a unique id. It can be chosen freely by the administrator. The engineID is included in the password hash algorithm. Therefore a message will only be accepted on the machine with the engineID that was used when creating the message. In other words: re-sending a captured message to another agent than the one, that was the receiver of the original message, will fail because, the password hash will be invalid.

Because it is still possible to re-send a request to the original target, the properties engineTime and engineBoots are used. EngineBoots holds the number reboots of an SNMP engine and engineTime the number of seconds that have passed since the last reboot of that engine. A client sending a GET or a SET request has to know the values of engineTime and engineBoots of the target agent and send them with the request. A receiving agent checks if the number of engineBoots is correct and if the value of engineTime does not differ more than 150 seconds from the current value of engineTime. This means, that means, that messages, that are re-send after the 150 seconds time window will not be accepted.

Which values for engineTime, engineBoots and engineID have to be used in a SNMP message depends on the type of the message. In the case of SET and GET messages the
target agent’s values have to be used. The agent is the authoritative SNMP engine for these requests, in SNMP jargon. When an agent is generating a response message to a GET or SET request it is still the authoritative engine. A message type where the engine initiating the communication is authoritative are TRAPs. TRAPs are asynchronous messages in SNMP usually sent from agents to clients to inform about some error situation or other event that has occurred. Engines receiving a TRAP do not generate a response of any kind. They may take specific actions when a TRAP is received, but there is no response or acknowledgment of any kind for the engine that has sent the TRAP. In contrast to TRAPs, GET and SET messages are synchronous. One request is always followed by exactly one response. Under certain conditions the response might be a REPORT message, which is presented in the next paragraph. A heuristic to decide which engine is authoritative in a communication is: "If I expect a response for my request, my communication partner is authoritative; otherwise I am."

![Figure 3.5: SNMPv3 time synchronization](image)

To be able to send requests with the correct values for engineTime, engineBoots and engineID, entities have to keep lists with these settings of their communication partners, constantly incrementing time. If an engine is communicating with a partner for the first time or if one of the settings has changed in a way that can not be predicted or calculated (e.g. an agent got rebooted because of an error situation), SNMP engines synchronize each other using a special REPORT message. Similar to TRAP messages, the sender of a REPORT is the authoritative engine and does not expect a response from the receiver. If an engine receives a request with invalid values for engineTime, engineBoots or engineID it generates an authentication failure REPORT, which, because itself is authoritative
for the REPORT, contains the current values for the triple. This is illustrated in figure 3.5. As mentioned earlier SNMP engines maintain a list of these values for the agents they have already communicated with. Using the values out of this list the second GET request from engine A to engine B does not generate a REPORT, because the values have been stored and updated by engine A and are therefore valid until the next reboot of engine B. The information an engine stores about engineBoots and engineTime of known communication partners, is updated every time a message is received from that partner, not only when receiving REPORTs. Note that a reboot in this context does not necessarily mean that the whole system has been rebooted. It just means, that the SNMP engine has for some reason been restarted.

Another security improvement over older SNMP versions, introduced with SNMPv3, is the fact, that passwords are no longer stored in clear text in configuration files. Similar to when passwords are transmitted in SNMP messages, they get encrypted for storage in a configuration file as well. One parameter used by the encryption algorithm is the engineID. Because of that, configuration files containing passwords are bound to a specific SNMP engine and become unusable once the engineID has changed. This is called password localization. To add a new user and password to an SNMP engine it is possible to set up the user name and his access rights in a clear text configuration file. The password itself and hash algorithm that should be used for this user has to be stored in clear text in another configuration file. After the SNMP engine is (re-)started for the next time it adds the new user and removes the clear text password, storing an encrypted one instead.

3.3.5 Multicast Mode for SNMP

Several of the communication concepts of SNMP communication, especially SNMPv3 communication, presented in the last section, cause problems when it comes to multicasting. This is not surprising as multicast communication was never intended when SNMP was created. This section presents the changes made to SNMP communication, gives details how the agents need to be configured and shows how the changes have been implemented in Net-SNMP.

![SNMP multicast communication diagram](image)

Because I wanted to apply as less changes as possible to SNMP, I’ve decided to use regular SNMP requests for GET and SET messages in multicast mode as well. This
means, that a the same message used for unicast communication is created for multicast communication as well, but it is send to a multicast IP address. As figure 3.6 shows, this message is not answered with a response message. The request and response pairs, used for unicast SNMP can not be used for multicast SNMP. Because there will be multiple agents (B and C) receiving the request and all of them will generate a response. But the requesting SNMP engine only expects one response and after that is received it will close the socket and all other responses will be lost. Therefore the requesting engine sends a request message and closes the socket right away, not expecting any response message. Instead, another application (A.2), running on the same machine as the requesting engine should receive the responses from all agents that have received the query. All received responses should then be forwarded to the requesting engine (A.1) using some form of interprocess communication. The responses should be send using an already existing message type: TRAPs. By using TRAPs as response messages it is possible to use Net-SNMP’s trap receiver service, named snmp-trapd, to listen for response TRAPs and to forward them to the requesting engine. In figure 3.6 the requesting engine is process A.1 and the receiving engine, snmp-trapd, is process A.2, which are running on the same machine.

This introduces several constraints regarding the agent’s configuration and problems, mostly related to SNMPv3:

- the very same request has to be successfully authenticated on all receiving agents
- agents must recognize this request as multicasted and answer with a TRAP rather than with a response message
- with the potentially large number of agents sending TRAPs, packet collisions should be avoided

When a request is multicasted to a group of agents is has to be accepted as a valid message by all agents in the multicast group. As only one request is generated and this one request is then delivered unchanged to all group members, all agents have to be set up with the same username and password which was used in the request. But even then there are problems. The parameters engineID, engineBoots and engineTime can not be the same on all receiving agents, by specification. Although, this means to break with the SNMP specification all agents need to be set up with the same engineID. Otherwise a request will only be authenticated on one agent in a multicast group. If all agents have the same engineID and the same username and password configured, a request will be considered valid by all agents regarding username, password and the SHA-1 and MD5 stream cipher. But, as all agents will be started and restarted independently engineTime and engineBoots will not be valid. I’ve decided to deactivate this setting completely. This means, that engineTime and engineBoots is always set to "0" and all agents will accept this value as valid. By considering this value as valid, agents will not generate any REPORTs regarding wrong values for engineTime and engineBoots. Although, this takes away some security it is also eliminating all communication overhead generated by REPORTs.

The next problem described above arises, after a request has been successfully authenticated and the handler function for the requested property is called by the Net-SNMP
agent. How can the handler function decide whether it should generate a response message for a unicast request or a TRAP for a multicast request?

As described in section 3.3.3, every handler function is called with the reqinfo structure as parameter. This structure contains the received SNMP data of the request, but not any UDP/IP data. Because of that it is not possible to check whether the received request was addressed to an unicast or a multicast IP address. But as figure 3.4 shows, the username used for the request is part of the SNMP request and therefore accessible through the reqinfo structure, which is a handle to the received request message. I’ve decided to use different usernames for unicast and multicast requests, which makes it possible to distinguish between unicast and multicast requests. All multicast requests have to be created using a username, that starts with the string "mc_". The rest of the username can be chosen freely. Unicast usernames can also be chosen freely with the exception that the first three characters must not be "mc_".

This makes it possible to decide what kind of message should be sent by a handler function (TRAP or response message). Response messages are generated by the Message Processing Subsystem of the Net-SNMP agent. Because of that the handler function does not need to know the IP address of the requesting host. The handler just sets the value that should be send in the response and the Message Processing Subsystem creates a SNMP response message for the requesting host. This does not work for TRAPs used to deliver results of requests. As mentioned in the previous section, TRAPs are usually used by SNMP agents to asynchronously report events to management stations. To which stations the TRAPs should be send to, is configured in the configuration file of the SNMP agent. If an event takes place, that causes a TRAP to be created by an agent, it will send the TRAP to all management stations listed in the configuration file. But in a scenario where there are multiple management stations performing management operations by multicasting requests to the agents running on the speaker systems, it is not desirable that agents send unicast TRAPs to all management stations. They should only be sent to the requesting management station. Otherwise this will create useless traffic.

I have therefore decided to introduce an additional configuration file for SNMP agents. In this configuration file there are all management stations listed with their IP address and a username. This means, that every management station has to perform multicast requests with its own dedicated username. Being able to get the IP address of the requesting management station via the one-to-one relation of username and IP address the handler functions of the custom MIB module of this project can build and send the TRAP message by themselves. As this behavior does not comply with the SNMP specification, according to which Net-SNMP’s agent is build there have to be some changes made to the agent as well.

After a handler function has finished it returns a status value, which gets interpreted by the Message Processing Subsystem, which is then creating a response message according to the status. Therefore I’ve added a new status value, called SNMP_TRAP_RESPONSE, which is returned when a TRAP has been successfully created and sent by a handler function. If that status is returned to the Message Processing Subsystem it will not create a response and mark the request as successfully handled.

TRAPs are received by a trap handler application. Trap handlers usually do not run on the same machines as agents, but often on the same machine as the SNMP client software,
the management station. As mentioned before, this is the case regarding the setup of this project (see figure 3.6). But not only the TRAPs used as responses to multicast requests might be sent by an agent, but also TRAPs informing about certain events, like for example if there is no space left on some memory device or authentication failures. This can be configured freely in the configuration of the SNMP agents. But a trap handler has to decide whether a received TRAP should be forwarded to another application because it is a response message or some other action has to be taken depending on the kind of event that has occurred.

I’ve added a new TRAP OID to the custom MIB file (see appendix B), which will be set by handler functions when they are generating TRAP responses. Net-SNMP’s trap handler `snmp_trapd` can then use this type to identify an incoming TRAP as a response, that should be forwarded. How the handling and forwarding is implemented is presented in detail in section 3.4.4.

To deal with the problem of collisions that might occur when a large number of agents send response TRAPs at almost the same time the property `delay`, part of the `sys-settings` group of the custom MIB module, which has already been presented in section 3.3.2 is used. This property sets up a timeframe of the property’s value in milliseconds, during which the agents send a TRAP. Similar to the behavior of nodes in an Ethernet network, when a collision has been detected, the agents choose a random number inside the timeframe after which they send their response TRAP. The value of the `delay` property is bound to a management station. This means, that requesting the value of this property from station X might lead to a different result than when the request is being sent from a station Y. This is necessary, because if there are multiple management stations in a network, they can choose their own timeframe, depending on how many agents are in the group they are currently sending requests to. The timeframe should be increased with the number of group members.

3.3.6 Summary

After the definition of a custom MIB module and the implementation of the handler functions for every property, controlling the various aspects of a speaker system was possible using Net-SNMP’s agent. But only unicast communication was possible. To enable multicast communication the handler functions needed to be extended, as well as several problems needed to be solved.

The following pseudo-code illustrates the behavior of a handler function in the custom MIB module that has been created for the project:

```
handler gets called
    decide what kind of operation has been requested (GET or SET)
    if this is a multicast request
        get the IP address for the response
        if the IP has been found
            perform the requested operation (GET or SET)
            select random number X between 0 and the value of delay
            sleep for X milliseconds
            send TRAP with the result of the operation
            return SNMP_TRAP_RESPONSE
```
else return SNMP_TRAP_RESPONSE (without sending a trap)

if this is a unicast request
    perform the requested operation (GET or SET)
    if the operation was successful
        update result variable
        return SNMP_SUCCESS
    else
        return SNMP_ERROR

With the presented changes a multicast communication with agents in the network is possible. The following list summarizes the performed changes:

• configure all agents with the same engineID

• disable checking for engineTime and engineBoots

• disable the generation of REPORT messages for invalid engineTime and engineBoots

• agents create multicast or unicast responses according to the username of the request

• agents send response TRAPs after a random time in a previously setup timeframe

• handler functions create response TRAPs and send it only to the requesting management station

• if a TRAP response has been sent, the Message Processing Subsystem does not send a response

• a new TRAP OID has been created to indicate response TRAPs

• unicast operation works according to the SNMP standard (with the exception of checking for engineID, engineBoots and engineTime)

3.4 Client GUI

The following sections discuss how the GUI was created. After an evaluation of the different cross-platform GUI frameworks presented in section 2.3, a general overview of the GUI is given as well as general usage information. The XML data model used to store configuration data for the speakers. The method used for interprocess communication with Net-SNMP’s trap receiver, used for responses to multicast SNMP requests, is described in detail as well.

3.4.1 Framework evaluation

As described in section 2.3, there have been three frameworks for cross-platform GUI creation taken into consideration. Out of these frameworks Qt was chosen for the creation of a client GUI. Qt provides several advantages over the two other solutions:

The lack of support for Apple’s Mac OS X by GTK+ is definitely a disadvantage compared to Qt and JFC. But the main reason for deciding against GTK+ was its
documentation and the organization of the project. The various parts of GTK+ reside on different websites and are organized in separate projects. For example, the Mac OS X port is not hosted as a part of the main GTK+ project. And also information about the status of each project does not always seem to be consistent. While the main project site claims that there is a Mac OS X port of GTK+, the actual website of the port says that there is a port in development and nothing has been released yet for uses any other than development. The documentation is divided into different parts as well, corresponding with the different parts of the GTK+ framework. But in general, the documentation is well written and seems to be up-to-date and complete.

Also the long list of dependencies on other libraries, like for example Cairo, does not make GTK+ a very convenient solution. Although the compilation of the GTK+ library itself usually is as simple as typing `configure`, `make`, and `make install` on a Linux console, you will have to install several libraries GTK+ depends on in the first place. Otherwise GTK+'s compilation will not succeed. It can be a time-consuming task to download and possibly compile and install the right combination of these libraries versions. Qt and JFC do not have these dependency issues. You download, maybe compile and install version x of these frameworks and you are done.

Qt and JFC provide good documentation, which is updated and released in sync with each new version of the frameworks. I have decided to use Qt to build the client console, because I prefer C++ over Java. Additionally Net-SNMP is written in C, so linking against libnetsnmp should cause less problems for a C++ application than for a Java application.

I have also made the experience, that although it is advertised, that you only need to write and compile an JFC application once to be able to use it on all supported platforms, that may not be true all the time. Depending on the version and vendor of the Java runtime on the target platform, a GUI’s appearance may differ. That means, that buttons are not in their proper positions and so on. With Qt all of my tests worked well on all supported platforms. A simple re-compile for the target platform produces the exact same result as on your development platform.

### 3.4.2 GUI Overview

Figure 3.7 shows a conceptual drawing of the client GUI. Based on drawings like this one, I've developed a GUI design with my supervisor at Bosch. The GUI is divided in three areas, shown in different colors in figure 3.7. All groups of speakers, that are currently set up are displayed in the green area, labeled "Group selection" in the image. All controls, which are related to group management are placed in this area, too. This means there are buttons, to let a user `add`, `remove` and `edit` groups.

The members of the selected group will be shown in the blue area, labeled "Main workspace". Using the controls in this part of the GUI, users can change the settings of the speakers, using either unicast or multicast messages. The buttons, to create and delete speakers are placed in the blue part of the GUI, too. The user should be able to choose between different views on the speakers. Views can be selected using the tabs in the blue area. Views include a table view, in which all speakers and their current settings are listed in a table. But also other views can be selected, like for example a Google Maps view, where the speakers are shown at their positions in a satellite image.

The yellow part, the "Toolbar" contains controls, to add speakers to or remove speakers
from groups, as well as buttons, to configure the client application and to upload files to speakers.

As shown in figure 3.8, which is a screenshot of the GUI’s implementation in Qt running on Linux, a status view has been added below the group and speaker views. Apart from the addition of the status log view, the GUI has been implemented as shown in the conceptual layout. Due to time constraints only the table view has been implemented, although the Google Maps view has been prepared by adding its tab to the GUI. The table contains one row per speaker and shows the settings for each property in the table’s columns. When a user changes a setting in a cell of the table, a dialog is presented to the user, asking, whether this value is to be set for the speaker to which the cell belongs or for all speakers currently shown in the table. If the user chooses to update only the current speaker a unicast SET request is generated and sent to the speaker. If the SET request is successful the new value is shown in the table, if not the previous value is kept. In any case the status is displayed in the status viewer.

If the user chooses to change the setting for all speakers, a multicast SET request is sent to the group of speakers, that is currently displayed in the table. To display the
status of the multicast query to the user a new dialog, shown in figure 3.9, is displayed. The dialog informs the user, about how many response TRAPs have not been received yet and how much time is left before the query times out. After either all responses have been received or the time is up, the user can close the dialog with the "Close" button. The table will then present the result of the query to the user. The cells displaying the value of the property will get colorized to provide a better overview. If a cell is colored green, this indicates that the request has been successful. This means, that the speaker has received the request and has send a response message, containing a success message. In addition to the green background of the cell, the value it displays is set to the requested value. A cell displaying a value, a speaker had previous to the request, with a yellow background indicates, that this speaker did not send a response message. There are two possible reasons for this. Either the speaker did not receive the request or we did not receive the response of the speaker. The user should then send a GET request to this speaker and thereby set the value displayed in the cell to the value currently set in the speaker. If a cell displaying the value previous to request has its background color set to red, the corresponding speaker did receive the request, but has reported that there has been an error setting the requested value. In this case the user might try to re-request the new value for this speaker or might try to restart the agent of this speaker.

By using different cell background colors, to indicate the result of a multicast query, the users gets an overview of the results very quickly without having to check if the values
are all actually set to the requested value. A quick scrolling through the table looking for colors other than green is enough.

If a user wants to update the value of a certain cell, it is enough to select the desired cell and hit the ”Get selected” button in the upper right corner. This will send a unicast GET request for the selected property of the selected speaker. If the request is successful the cell’s value is updated with the received value, otherwise an error message is displayed to the user. If the user would like to update a specific value for all speakers in the group, that is currently displayed, he/she has to click on the ”Get group” button instead. This will send a GET request for the selected property to the multicast group address of the currently displayed group. Like when setting values with a multicast request the window shown in figure 3.9 is shown, informing the user about the progress of the query. After all results have been received or the request has timed out the table gets updated, too. Colors are used to indicate the results of a query in the same way as after SET requests. But there is one exception. GET requests always either succeed or time out. Therefore, cells can either be presented with green background, indicating that the current value has been successfully received or yellow background, indicating, that no response at all has been received.

The button labeled ”Settings...”, in the toolbar area, presents a user with a dialog to configure the client GUI. Values for the usernames and passwords for unicast and multicast communication can be set, as well as timeouts for unicast and multicast requests. Next to the ”Settings...” button, is a button with the label ”Upload...”. Clicking on this button will display a new dialog window, that lets the user choose a file for upload to a speaker. The upload is done using the scp shell command via a secure SSH connection to the target speaker.

As figure 3.8 shows, there are also settings shown in the table that are not properties of the custom MIB module. Namely those are:

- name: Hostname of a speaker
- mac: MAC address of a speaker
- ip: IP address of a speaker
- location: Location of a speaker
• long: Longitude of a speaker’s position (for Google Maps view)

• lat: Latitude of a speaker’s position (for Google Maps view)

All of these values except long and lat are available for remote management by the MIB-II module, which every SNMP application has to implement. The long and lat properties are preparations for the Google Maps view mentioned earlier. If this view is implemented at a later time, I would suggest to add these values to a custom MIB module. However, at this time all of these properties are managed only locally in the client GUI. This means, that their setting is not related to any property on the speaker and no SNMP messages are sent when their values get changed. Due to time constraints it was not possible to change the MIB-II module to work with the multicast extension for SNMP, presented in this paper.

When a user wants to add a new speaker to the list, he/she clicks on the “+” button in the upper right corner of the GUI and a new dialog is displayed. In the fields of this dialog, a user has to enter all non-MIB values (name, mac, ip, location, long and lat) of the speaker that he/she wants to add. After that all values are fetched by the GUI using GET requests.

3.4.3 XML Data model

In figure 3.8 there is a button called ”File” in the menu bar of the window. This button displays, when it is clicked, a standard file menu known from most applications. It displays the entries ”New”, ”Close”, ”Open...”, ”Save”, ”Save as...” and ”Quit” to the user. Selecting ”Quit” will exit the application. All other options relate to file operations. It is possible to save the current setup of speakers and groups in a file and load it again at a later time. This makes it very convenient to move configuration between different computers running the client GUI.

The settings are stored in an XML-based file. A file containing a speaker and a group might look like this:

```xml
<speakerList>
  <speaker>
    <name>agent1</name>
    <mac>00:0e:99:02:07:8d</mac>
    <ip>192.168.0.2</ip>
    <location>2nd floor</location>
    <long>5</long>
    <lat>5</lat>
    <volume>76</volume>
    <reload>0</reload>
    <reboot>0</reboot>
    <exec>0</exec>
    ...
    <maxVolume>100</maxVolume>
  </speaker>
  <group name="group1" ip="224.0.0.11"/>
</speakerList>
```
Note that most settings of the speaker are left out in this example. All properties that are defined for a speaker are stored in the XML file with a tag name equal to the name of the property.

A group tag has two attributes: name and ip. The name attribute represents the name of the group shown in the GUI and the attribute ip represents the multicast IP address of the group. All speakers, that are configured with this multicast address, using the mgroup property, belong to this group.

The GUI is using the Document Object Model (DOM) to handle XML documents. Using DOM a complete copy of an XML file is loaded into memory when the file is parsed. The GUI is using this XML document as a data model. This means, that all changes or updates, that are made to speakers and groups are made directly in the XML document. The GUI is always displaying the values in the XML document. If the user selects for example a certain group in the group selection list on the left side of the GUI, the table will be reloaded going through the XML document in memory and adding only those speakers to the table that are in the currently selected group.

When the user selects to save the current settings, the XML document in memory is written to a file again.

3.4.4 Communication with the trap receiver

When an agent is receiving a request, which it identifies as multicast request, as described in section 3.3.5, it will not send a response message, but a TRAP to the requesting management station. I wanted to use Net-SNMP’s trap receiver snmp_trapd, to receive the TRAPs generated by agents, when answering to multicast requests. When TRAPs are sent, the sender is authoritative (see section 3.3.4). This means, that TRAPs contain the engineID of the sender and the engineID of every authorized sender has to be configured in snmp_trapd’s configuration file. Along with the engineID a username, password and wether SHA-1 or MD5 is used has to be specified in the configuration file. After this data has been entered snmp_trapd does accept TRAPs from this sender.

When a TRAP has been accepted by the trap receiver it will by default add the data contained in the TRAP to its log file. But it is possible to define traphandles, which can be used to let the receiver perform user defined operations upon reception of certain TRAPs. To let snmp_trapd execute the handle_trap shell script of the client GUI the following line has been added to its configuration file:

```
traphandle OMAP-MIB::trapResponse /<path_to_gui>/handle_trap
```

Note that the custom MIB file for this project, called OMAP-MIB contains a special TRAP type (see section 3.3.5 and appendix B), used for TRAPs containing responses for multicast requests. This special TRAP type is named trapResponse. The above line causes snmp_trapd to execute the shell script /<path_to_gui>/handle_trap when a TRAP of type trapResponse is received. The reaction for other TRAP types is not affected by this setting.

The shell script handle_trap, which gets called for TRAPs of the type trapResponse looks like this:

```
#!/bin/bash
```

59
The shell scripts first read the arguments host and orig_ip which have been added by snmp_trapd as arguments when executing the shell script. Several OID and value pairs have been added as parameters as well. The first OID defines the type of this TRAP (trapResponse) and the second one defines a time stamp. The last OID defines the requested property and its value. The script cycles through all OIDs and stores the last one and its value. Having the OID-value pair for the requested property the script extracts the IP address of the sender out of the string "<protocol>:<ip>:<port>". After all data is gathered and well formatted, the script tests, if there exists a named pipe in the current directory (the directory where the GUI is installed), named trap_pipe. If the pipe exists, it writes IP, OID and value separated by the character "!" into the pipe. If the pipe does not exist, it just logs the three values to standard output.

A named pipe is a mechanism used for interprocess communication. A named pipe

Figure 3.10: Communication between trap receiver and client GUI
acts like a First In First Out (FIFO) buffer and has two "ends". This means, that one process opens the pipe for reading (the reading end) and the second process opens the pipe for writing (the write end). The process at the writing end can only write to the pipe and the process on the reading end can only read from the pipe. Everything written to the pipe can be read by the other process on the reading end.

The GUI is waiting at the reading end of the pipe for information about received TRAPs. This information is written by the above shell script into the pipe. Figure 3.10 illustrates this behavior. Because a process is blocked, when it is waiting at the reading end of an empty pipe until input is written to the pipe, I have created a thread, called pipeReader. The pipeReader thread is started when the GUI application is started and is always trying to read data from the pipe. If data has been successfully read from the pipe it emits a SIGNAL with the data, that has been read. SIGNALs and SLOTs are a mechanism provided by Qt for event handling. When a SLOT is connected to a SIGNAL, the SLOT function is called every time the SIGNAL is emitted. It is possible to dynamically connect and disconnect SIGNALs and SLOTs at runtime.

When the user has selected to send a multicast query, the dialog shown in figure 3.9, running in the main thread is shown and a multicast request is send (see figure 3.10). This dialog connects a SLOT to the SIGNAL, that is emitted by the pipeReader upon reception of a TRAP. When the agents send their TRAP responses, they are received by the snmp_trapd, which is calling the handle_trap script. The script is writing the data of the TRAP to the named pipe, where it is read by the pipeReader thread of the Client GUI. If the data is valid, the pipeReader emits a SIGNAL, which is received by the multicast dialog in the main thread. After all responses have been received or the query timed out, the multicast dialog will disconnect its SLOT from pipeReader’s SIGNAL, therefore not receiving information about received traps anymore.

Because the pipeReader thread is always running and reading from the pipe, there is no risk, that it will get "full". The amount of data that can be stored in a named pipe is limited and the behavior of the operating system once a pipe is full is not specified by POSIX and may differ between different implementations.

3.4.5 Summary

A Client GUI has been created using Qt as cross-platform framework for GUI development. The GUI is communicating with the snmp_trapd, which is used to receive the incoming TRAPs from the agents, using a named pipe. To be able to read all incoming data out of the pipe at all times, a thread (pipeReader) has been added to the GUI. The pipeReader thread is always trying to read from the pipe and emits a Qt SIGNAL upon success. Once a multicast query has been started, the main thread connects a SLOT to the SIGNAL of the pipeReader. All data read from the pipe is then forwarded by the SIGNAL to the main thread. After the multicast query is finished, the main thread disconnects again, because it is no longer interested in incoming TRAP data. Although the data is not being used, once the main thread has disconnected, the pipeReader is still reading it. This prohibits the pipe from running out of space.
4 Results

This section presents the results of the different parts of the project. It also shows several benchmarks and user opinions.

4.1 Development environment

The development environment provides a menu-driven way to configure toolchain, kernel and filesystem images for the target system. The development environment is based on the Buildroot open-source project. It has been extended with several new software packages. Some bugfixes mostly related to boot scripts and configuration files have been added as well.

A major new feature, that has been added to Buildroot, is the possibility to perform kernel builds. There has also been a lot of effort put into improving the support of the special hardware of the embedded board and Wireless LAN support.

The tool has proven itself very useful and reliable to me while I was developing the management interface and I have received positive feedback from colleagues. Bosch is planning to use the development environment to enable students to get started with development of embedded applications more quickly in the future. There has also been interest from another department in the tool, for another project with ARM-based embedded hardware.

4.2 Management interface

From all presented management solutions, SNMP was the only one providing UDP based transport. Additionally to this advantage, the Net-SNMP open-source implementation also has very good properties regarding the use in an embedded environment. I’ve therefore implemented a management interface for various system-related settings as well as the stream-related settings using Net-SNMP. The MIB compiler provided by Net-SNMP has been a useful tool to speed up the implementation of the custom MIB module.

In addition to the management interface itself, a lot of work has been put into making the multicasting of SNMP requests possible. By using multicast the major goal was to keep response times of multiple agents short, but also to reduce the overall bandwidth consumption of the traffic created by management operations.

I have recorded several sets of test values to compare unicast SNMP communication with multicast communication. Due to the lack of enough WiFi-Adaptors I’ve been using a 100 MBit/s wired Ethernet connection to record the test values. I’ve connected five embedded agents and one management console to perform two test scenarios. In one scenario the current setting of an integer value should be retrieved form the agents. This GET request has been send using unicast communication as well as with multicast. Figure 4.1 shows the results. The chart shows the average values of 10 recorded values. Values have been recorded for retrieving a value from one to five agents. Once when there was no other traffic on the network and another time while the agents were playing back a live audio stream. Note that SNMPv3 has been used for unicast communication, considering the best possible case, where engineID, engineBoots and engineTime are known to the management station. This means that there is no additional delay and traffic generated by REPORTs.
The unicast requests have been sent using a loop function. This means, that the request to the second agent is sent, once the response of the first agent has been received. When the first request is out on the network the clock was started and once the last response has been received the clock was stopped. For the multicast tests, the agents were using a 100 milliseconds timeframe for their response. Response times could even improve when a smaller delay time frame is chosen for this small amount of agents.

The recorded values show, that unicast communication produces superior values when a very small number of agents are target of a request. I think this is mainly due to the more complex operations that the agents are performing when answering to a multicast response. But with increasing number of agents multicast produces better results. As said before, the time needed for example to forward received TRAP responses to the management application has not been recorded in these tests. As figure 4.1 shows response times scale a lot better with an increasing number of agents when multicast communication is used.

<table>
<thead>
<tr>
<th>Operation</th>
<th>1 agent</th>
<th>2 agents</th>
<th>3 agents</th>
<th>4 agents</th>
<th>5 agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unicast GET (int)</td>
<td>332</td>
<td>664</td>
<td>996</td>
<td>1328</td>
<td>1660</td>
</tr>
<tr>
<td>Multicast GET (int)</td>
<td>368</td>
<td>571</td>
<td>774</td>
<td>977</td>
<td>1180</td>
</tr>
<tr>
<td>Unicast SET (char[6])</td>
<td>347</td>
<td>694</td>
<td>1041</td>
<td>1388</td>
<td>1735</td>
</tr>
<tr>
<td>Multicast SET (char[6])</td>
<td>376</td>
<td>579</td>
<td>782</td>
<td>985</td>
<td>1188</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of Bytes transfered when using unicast or multicast
Multicast communication is also performing better regarding the amount of data that has to be sent over the network to complete these requests. As table 4.1 shows, unicast has a slight advantage when only one agent is queried. This is because data sent using a TRAP is 29 Bytes bigger than when sent using a response. This is due to the time-stamp the TRAP contains. As more agents are queried the gap between multicast and unicast is getting bigger and bigger. When for example five agents are target of a SET request for a six character long string, there will be 1735 Bytes using unicast, compared to 1188 Bytes using multicast (see table 4.1). All figures presented in table 4.1 are not considering any REPORTs and subsequent re-sending of requests, that might occur when using unicast SNMPv3. The figures represent the best-case scenario.

Figure 4.2: Response times of SET request for a 7 character string

Figure 4.1 shows the response times for a GET request, figure 4.2 shows the response times of a SET request. For the GET request an integer type property has been requested. The SET request contains a six character long string. Because in unicast communication the responses to SET requests contain the requested value and TRAP responses include only a success identifier, the response TRAPs are smaller than the responses in this case. In all other aspects the test setup has been identical with described setup for the GET test. Again, time has been measured without any traffic on the network, as well as while the agents are receiving a live audio stream. Similar to the results of the previous test, unicast is faster than multicast for requests targeting a small number of agents. When more than three agents are queried, multicast is faster than unicast and it is scaling better with every additional agent being queried.

I think because of that numbers the advantages gained in bandwidth consumption and response times, outweigh the disadvantages in security, that come with the changes.
Although the possibility of replay attacks is possible now and is by itself a serious problem, the wireless network is secured by WPA. Therefore the risk of someone recording SNMP packets with a packet sniffer is very low, as only authorized personnel has access to the network in the first place and through WPA all packets are encrypted by AES or TKIP, depending on the used WPA version.

4.3 Client GUI

The management application that has been created as a client for the management interface supported me well, while performing the tests described above. However, the GUI is a proof-of-concept work and is in some regards to complicated and inconvenient for the end-user. For example the creation of scheduled events needs to be done by hand. This means, that a user has to enter a string like "a:00:22:*:*:1-5:/closingTime.mp3" into the "recurring" cell to add an event. It would be much more useable, when there would be some calendar presented to the user where he can choose data and time for the event in a way that feels more natural to most users.

But the general idea of a table to view the settings of all speakers has proven to be good. For example you can see quickly, after creating a new group, where there are differences in the setup of the group members. Also the coloring of the cells, to indicate the result of a multicast request has proven itself very useful and has created positive feedback from colleagues.

The concept of using a named pipe for communication with the trap receiver has turned out to be very reliable and has not ever caused any problems. One problem related with this method, though is the fact that the GUI does not run on Windows. Although GUI is technically still running on Windows, the concept of named pipes is not available there in the same way as on a UNIX system. Therefore no TRAP responses to multicast requests will be received by the GUI when running on Windows. However, a similar mechanism is available on Windows, too. So porting the application to Windows will take little effort.

The GUI is a useful tool to manage speakers and is very reliable in its operation. However, it is neither intended to be used by an end-user, nor is it ready for it. Changes regarding the usability need to be made, hiding several unnecessary complexities from the user. Although, the GUI is running flawlessly on Linux and Mac OS X, changes need to be made to support Windows as well.
5 Conclusion and future work

Besides the objectives of this project there are several possible extensions and improvements to the presented system. This section describes them, as well as conclusions from the project work.

5.1 Future work

5.1.1 Database storing speaker data

The usage of XML as data model for speaker data is working very well with that rather small number of speakers, that need to be managed. But if the number of speakers increases to several hundreds, there might be performance problems introduced by the operations on the DOM document. These might get slow and the amount of memory required to hold the DOM document in memory might cause problems. I would therefore suggest to use a database as data model. This should solve both problems, which might be caused by XML and a high number of speakers. Compared to the amount of data a database is developed for, the data of even more than a thousand speakers is very small.

5.1.2 Multicast response times

Although, the performed tests, show an significant advantage in using the developed multicast communication over the standard unicast communication there is still some space for improvements left. It would be an interesting test to see if there could be additional performance gains when a more powerful hardware is used for the speaker systems. In my opinion most of the faster response times using unicast when a small number of agents is queried, is due to the more complex computations done in multicast mode.

5.1.3 Bandwidth consumption

Bandwidth consumption and therefore the time needed to deliver a packet, could be further reduced if the size of the response TRAPs is further reduced. Although this would introduce an additional difference to the SNMP standard, the time-stamp could be removed from the response TRAPs. This would for example save 29 Bytes per response for a GET request. This means, that when querying 100 agents, 2900 Bytes less would need to be send.

Another possibility to reduce network traffic is related to response TRAPs as well. Think of a system where there are more than one management station. A way to reduce traffic in such a setup, is to reduce the number of polls made from management stations to agents. Say station A requests a value from some agents. In the presented solution, theses agents would respond with unicast messages to this agent. If the responses would be multicasted to all other management stations as well, the amount of polling for the same values could be reduced significantly. This means, that every station profits from a poll, not only the one performing the poll.

5.2 Conclusion

Throughout the seven months of work on the project most of the objectives that have been set could be accomplished. A menu-driven development environment for the target’s
OMAP hardware on the base of the Buildroot system has been created. The development system’s build system performs automated builds of kernel images, applications and filesystem images. In these builds a cross-compiler toolchain is used, which has been build by the development environment as well. It is automatically applying patches necessary for the OMAP board’s hardware and keeps track of dependancies between different software packages.

On the side of the management interface a lot of work has been done as well. After getting to know several different management solutions and their properties, I’ve done an installation of most of them to check whether they fit into an embedded environment or not. With some applications my experiences regarding memory consumption differ significantly with the claims of the manufacturer of the software. However, SNMP was the only solution providing communication over the connectionless UDP protocol, thereby having the possibility to use multicast. After research regarding the SNMP protocol, I found out that the use of multicast was never considered when the protocol was created. In fact, several properties of the protocol prohibit a successful multicasting of messages.

Because of that I was looking for a way to make as less changes as possible to the standard protocol to make multicast work. The solution of using TRAPs as response messages has proven itself as reliable. Also the changes to the SNMP agent’s implementation where rather small. A new status type `SNMP_TRAP_RESPONSE` has been created to indicate that a response has already been sent, to the Message Processing Subsystem of the agent. A mechanism for creating and sending response TRAPs has been created. This mechanism is using a pre-defined delay to avoid package collisions when querying a large number of agents.

The custom MIB module created to make the settings specific to the project’s speaker system manageable has been implemented for the Net-SNMP agent. Although currently all required aspects of the speaker are manageable with the MIB module and future extensions, like acoustic guidance, have already been prepared, it is likely that there have to be some domain specific additions to the MIB module for an actual product. Changes and additions to the module are no problem at all, a simple re-compile of the agent is necessary after the changes have been made in the source code.

A domain-specific adaption of the Client GUI will also be necessary for an actual product. Aside from several complexities, that should be hidden from the user, the GUI is a very reliable solution to manage a speaker’s settings. With the use of Qt it is already running on Linux and Mac OS X. Windows support can be added easily, by using the Windows named pipe system instead of the UNIX one. All other aspects of the GUI are already supporting Windows.

Overall, the objectives of the project have been accomplished. Although there have to be more tests proving the systems response times to be fast enough with several hundreds of agents, which was not possible due to the constraints regarding the available hardware.
References


A Glossary

API (Application Programming Interface) An API offers an interface to developers, to support them by using functions of a library or system.

ARM (Acorn RISC Machine) An CPU core design for a family of microprocessors by the company ARM.

DES (Data Encryption Standard) A wide spread symmetric encryption algorithm.

DSP (Digital Signal Processor) A computer processor, specialized for the processing of digital signals.

GUI (Graphical User Interface) The part of a program, that enables the user to control the program and input data with a mouse and keyboard.

Java Highly object-oriented programming language, developed by Sun Microsystems.

JFFS2 (Journaling Flash Filesystem v2) Filesystem for use in flash memory devices.

MD5 (Message-Digest Algorithm 5) A wide spread cryptographic hash function, used to calculate checksums.

MMU (Memory Management Unit) The part of a computer, responsible for keeping track of the computer’s memory and the access to it.

PMS (Package Management System) An application used to manage software packages for an installation.

POSIX (Portable Operating System Interface) International standard for the UNIX API.

RPC (Remote Procedure Call) Protocol for distributed applications.

SCP (Secure Copy) SCP is a program as well as a protocol for encrypted file transfers over a network.

SHA (Secure Hash Algorithm) SHA refers to a group of cryptographic hash functions, used to create a unique checksum for arbitrary electronic data.

SSH (Secure Shell) Secure shell or SSH refers to a protocol as well as a program for secure remote logins between networked computers.

SUSv3 (Single Unix Specification v3) Specification of the UNIX API, similar to POSIX.

TCP (Transmission Control Protocol) Connection oriented protocol often used in conjunction with IP protocol as base for various network services.

TFTP (Trivial File Transfer Protocol) A very simple file transfer protocol, supporting reading and writing of files over a network.
TTL (Time to Live) One field of the header of the IP protocol, which prevents undeliverable packets from being routed to infinity.

WPA (WiFi Protected Access) Standard for protecting access to wireless networks, providing secure authentication and privacy.

X11 (X Version 11) System to display GUIs on UNIX systems and OpenVMS.

XML (Extensible Markup Language) Markup language used to structure different kinds of informations and to define new document formats. XML files are human-readable and consist of text. [http://www.w3.org/XML](http://www.w3.org/XML)
B OMAP-MIB

OMAP-MIB DEFINITIONS ::= BEGIN

IMPORTS
enterprises, IpAddress, Integer32,
MODULE-IDENTITY FROM SNMPv2-SMI
DateAndTime, DisplayString FROM SNMPv2-TC
TRAP-TYPE FROM RFC-1215
MODULE-COMPLIANCE, OBJECT-GROUP FROM SNMPv2-CONF;

bosch OBJECT IDENTIFIER ::= { enterprises 5318 }

omap MODULE-IDENTITY
  LAST-UPDATED "1004080000Z"
  ORGANIZATION "Robert Bosch GmbH, Corporate Research"
  CONTACT-INFO "external.andre.eisenmann@de.bosch.com"
  DESCRIPTION "Test MIB for system/stream management"
  ::= { bosch 1 }

sys-settings OBJECT IDENTIFIER ::= { omap 1 }
stream1 OBJECT IDENTIFIER ::= { omap 2 }
stream2 OBJECT IDENTIFIER ::= { omap 3 }
stream3 OBJECT IDENTIFIER ::= { omap 4 }
stream4 OBJECT IDENTIFIER ::= { omap 5 }
playback OBJECT IDENTIFIER ::= { omap 6 }
aGuidance OBJECT IDENTIFIER ::= { omap 7 }
traps OBJECT IDENTIFIER ::= { omap 8 }

--------------------------------------------
--SYSTEM PART--
--------------------------------------------

-- main volume
volume OBJECT-TYPE
  SYNTAX Integer32 (0..100)
  ACCESS read-write
  STATUS mandatory
  DESCRIPTION "main playback volume"
  DEFVAL { 76 }
  ::= { sys-settings 1 }

-- snmpd reload-config
reload OBJECT-TYPE
SYNTAX Integer32(0..1)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"'true' will cause snmpd to reload its config"
DEFVAL { false }
 ::= { sys-settings 2 }

-- reboot system
reboot OBJECT-TYPE
SYNTAX Integer32(0..1)
ACCESS read-write
STATUS mandatory
DESCRIPTION
"'true' will cause a system reboot"
DEFVAL { false }
 ::= { sys-settings 3 }

-- execute arbitrary command
exec OBJECT-TYPE
SYNTAX DisplayString (SIZE (0..50))
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"send shell command to be executed by the agent"
 ::= { sys-settings 4 }

-- IP of management group
mgroup OBJECT-TYPE
SYNTAX IpAddress
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"IP address of the management group this speaker belongs to"
 ::= { sys-settings 5 }

-- set/get delay
delay OBJECT-TYPE
SYNTAX Integer32(1..999)
ACCESS read-write
STATUS mandatory
DESCRIPTION
"Delay for trap responses in milliseconds"
DEFVAL { 0 }
 ::= { sys-settings 6 }
-- enable/disable ssh access on the target
upload OBJECT-TYPE
SYNTAX Integer32(0..1)
ACCESS read-write
STATUS mandatory
DESCRIPTION
"'1' enables, '0'disables SSH access to the agent"
DEFVAL { 0 }
 ::= { sys-settings 7 }

-- STREAMING PART --

-- Name or description of stream1
streamName1 OBJECT-TYPE
SYNTAX DisplayString(SIZE(0..30))
ACCESS read-write
STATUS current
DESCRIPTION
"Name or description of stream #1"
 ::= { stream1 1 }

-- IP adress to receive a stream from
ipAddr1 OBJECT-TYPE
SYNTAX IpAddress
ACCESS read-write
STATUS current
DESCRIPTION
"IPv4-address to receive a stream from"
 ::= { stream1 2 }

-- port adress to receive a stream from
portNum1 OBJECT-TYPE
SYNTAX Integer32(1..50000)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Port to receive a stream from"
DEFVAL { 0 }
 ::= { stream1 3 }

-- volume modifier for stream 1
volumeMod1 OBJECT-TYPE
SYNTAX Integer32(-100..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"volume modifier for stream #1"
DEFVAL { 0 }
::= { stream1 4 }

-- delay for stream 1
delay1 OBJECT-TYPE
SYNTAX Integer32(0..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"delay for stream #1 in milliseconds"
DEFVAL { 0 }
::= { stream1 5 }

-- activate stream 1
activate1 OBJECT-TYPE
SYNTAX Integer32(0..1)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"set to 1 to activate stream1, 0 deactivates"
DEFVAL { 0 }
::= { stream1 6 }

-- STREAM2 --
-- Name or description of stream2
streamName2 OBJECT-TYPE
SYNTAX DisplayString(SIZE(0..30))
ACCESS read-write
STATUS current
DESCRIPTION
"Name or description of stream #2"
::= { stream2 1 }

-- IP address to receive a stream from
ipAddr2 OBJECT-TYPE
SYNTAX IpAddress
ACCESS read-write
STATUS current
DESCRIPTION
"IPv4-address to receive a stream from"
::= { stream2 2 }
-- port adress to receive a stream from
portNum2 OBJECT-TYPE
SYNTAX Integer32(1..50000)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Port to receive a stream from"
DEFVAL { 0 }
::= { stream2 3 }

-- volume modifier for stream 2
volumeMod2 OBJECT-TYPE
SYNTAX Integer32(-100..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"volume modifier for stream #2"
DEFVAL { 0 }
::= { stream2 4 }

-- delay for stream 2
delay2 OBJECT-TYPE
SYNTAX Integer32(0..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"delay for stream #2 in milliseconds"
DEFVAL { 0 }
::= { stream2 5 }

-- activate stream 2
activate2 OBJECT-TYPE
SYNTAX Integer32(0..1)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"set to 1 to activate stream2, 0 deactivates"
DEFVAL { 0 }
::= { stream2 6 }

-- STREAM3 --
-- Name or description of stream3
streamName3 OBJECT-TYPE
SYNTAX DisplayString(SIZE(0..30))
ACCESS read-write
STATUS current
DESCRIPTION
"Name or description of stream #3"
 ::= { stream3 1 }

-- IP address to receive a stream from
ipAddr3 OBJECT-TYPE
SYNTAX IpAddress
ACCESS read-write
STATUS current
DESCRIPTION
"IPv4-address to receive a stream from"
 ::= { stream3 2 }

-- port address to receive a stream from
portNum3 OBJECT-TYPE
SYNTAX Integer32(1..50000)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Port to receive a stream from"
DEFVAL { 0 }
 ::= { stream3 3 }

-- volume modifier for stream 3
volumeMod3 OBJECT-TYPE
SYNTAX Integer32(-100..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"volume modifier for stream 3"
DEFVAL { 0 }
 ::= { stream3 4 }

-- delay for stream 3
delay3 OBJECT-TYPE
SYNTAX Integer32(0..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"delay for stream #3 in milliseconds"
DEFVAL { 0 }
 ::= { stream3 5 }

-- activate stream 3
activate3 OBJECT-TYPE
SYNTAX Integer32(0..1)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"set to 1 to activate stream3, 0 deactivates"
DEFVAL { 0 }
 ::= { stream3 6 }

-- STREAM4 --
-- Name or description of stream4
streamName4 OBJECT-TYPE
SYNTAX DisplayString(SIZE(0..30))
ACCESS read-write
STATUS current
DESCRIPTION
"Name or description of stream #4"
 ::= { stream4 1 }

-- IP address to receive a stream from
ipAddr4 OBJECT-TYPE
SYNTAX IpAddress
ACCESS read-write
STATUS current
DESCRIPTION
"IPv4-address to receive a stream from"
 ::= { stream4 2 }

-- port address to receive a stream from
portNum4 OBJECT-TYPE
SYNTAX Integer32(1..50000)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Port to receive a stream from"
DEFVAL { 0 }
 ::= { stream4 3 }

-- volume modifier for stream 4
volumeMod4 OBJECT-TYPE
SYNTAX Integer32(-100..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"volume modifier for stream #4"
DEFVAL { 0 }
 ::= { stream4 4 }
-- delay for stream 4
delay4 OBJECT-TYPE
SYNTAX Integer32(0..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"delay for stream #4 in milliseconds"
DEFVAL { 0 }
::= { stream4 5 }

-- activate stream 4
activate4 OBJECT-TYPE
SYNTAX Integer32(0..1)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"set to 1 to activate stream4, 0 deactivates"
DEFVAL { 0 }
::= { stream4 6 }

--- PLAYBACK PART ---

-- file to play
filename OBJECT-TYPE
SYNTAX DisplayString (SIZE (0..50))
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Absolute path and filename of the file to play"
DEFVAL { 0 }
::= { playback 1 }

-- define a time for playback
pTime OBJECT-TYPE
SYNTAX DisplayString(SIZE(1..8))
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Define a time for playback to start. Format: MMDDhhmm.
Setting 1, will start playback right away."
DEFVAL { 0 }
::= { playback 2 }
-- add/remove auto-play event, "return value" will tell if successful recurring OBJECT-TYPE
SYNTAX DisplayString(SIZE(0..50))
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"[a|d]:mm:hh:DD:MM:WeekDay:FileName
Add or delete a file for automatic local playback"
DEFVAL { 0 }
::= { playback 3 }

-- add/remove auto-play event, "return value" will tell if successful
listEvents OBJECT-TYPE
SYNTAX DisplayString(SIZE(1..50))
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Sending n will give you information about auto-play event n.
If n events are scheduled, a query for m (m>n) will return NONE."
DEFVAL { 0 }
::= { playback 4 }

-- check if all files are available
fileCheck OBJECT-TYPE
SYNTAX Integer32(0..99)
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"Check presence of all files that are scheduled for playback.
Variable will be set to 1 if all files are present. Otherwise 0"
DEFVAL { 0 }
::= { playback 5 }

--------------------------------------------
-- ACOUSTIC GUIDANCE PART --
--------------------------------------------

-- start time --
startTime OBJECT-TYPE
SYNTAX DisplayString SIZE(8)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Define a time for playback to start. Format: MMDDHHMM.
Setting 1, will start playback right away."
DEFVAL { 0 }
::= { aGuidance 1 }

-- start delay (t0)
startDelay OBJECT-TYPE
SYNTAX Integer32(0..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Define a start delay"
DEFVAL { 0 }
::= { aGuidance 2 }

-- start ramp up(t1)
startRampUp OBJECT-TYPE
SYNTAX Integer32(0..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Define start ramp up"
DEFVAL { 0 }
::= { aGuidance 3 }

-- stop ramp up(t2)
stopRampUp OBJECT-TYPE
SYNTAX Integer32(0..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Define stop ramp up"
DEFVAL { 0 }
::= { aGuidance 4 }

-- start ramp down(t3)
startRampDown OBJECT-TYPE
SYNTAX Integer32(0..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Define start ramp down"
DEFVAL { 0 }
::= { aGuidance 5 }

-- stop ramp down(t4)
stopRampDown OBJECT-TYPE
SYNTAX Integer32(0..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Define stop ramp down"
DEFVAL { 0 }
::= { aGuidance 6 }

-- loop time(t5)
loopTime OBJECT-TYPE
SYNTAX Integer32(0..1000000)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Define length of the whole loop"
DEFVAL { 0 }
::= { aGuidance 7 }

-- min. volume(t0-t1 and t4-t5)
minVolume OBJECT-TYPE
SYNTAX Integer32(0..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Define minimal volume"
DEFVAL { 0 }
::= { aGuidance 8 }

-- max. volume(t2-t3)
maxVolume OBJECT-TYPE
SYNTAX Integer32(0..100)
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"Define maximal volume"
DEFVAL { 0 }
::= { aGuidance 9 }

trapResponse TRAP-TYPE
ENTERPRISE traps
DESCRIPTION
"Used as response for Multicast requests"
::= 1

END
C  Mib2c example files

File: OMAP-MIB.txt

OMAP-MIB DEFINITIONS ::= BEGIN

IMPORTS
enterprises, Integer32,
MODULE-IDENTITY FROM SNMPv2-SMI
MODULE-COMPLIANCE, OBJECT-GROUP FROM SNMPv2-CONF;

bosch OBJECT IDENTIFIER ::= { enterprises 5318 }

omap MODULE-IDENTITY
   LAST-UPDATED "1004080000Z"
   ORGANIZATION "Robert Bosch GmbH, Corporate Research"
   CONTACT-INFO "external.andre.eisenmann@de.bosch.com"
   DESCRIPTION "Test MIB for system/stream management"
   ::= { bosch 1 }

sys-settings OBJECT IDENTIFIER ::= { omap 1 }

-- main volume
volume OBJECT-TYPE
   SYNTAX Integer32 (0..100)
   ACCESS read-write
   STATUS mandatory
   DESCRIPTION "main playback volume"
   DEFVAL { 76 }
   ::= { sys-settings 1 }

END

File: omap.h

/*
 * Note: this file originally auto-generated by mib2c using
 *       : mib2c.scalar.conf,v 1.9 2005/01/07 09:37:18 dts12 Exp $
 */
#ifndef OMAP_H
#define OMAP_H

/*
 * function declarations
 */
void init_omap(void);
Netsnmp_Node_Handler handle_volume;

84
```c
#include <net-snmp/net-snmp-config.h>
#include <net-snmp/net-snmp-includes.h>
#include <net-snmp/agent/net-snmp-agent-includes.h>
#include "omap.h"

/** Initializes the omap module */
void
init_omap(void)
{
    static oid volume_oid[] = { 1, 3, 6, 1, 4, 1, 5318, 1, 1, 1 }
;
    DEBUGMSGTL(("omap", "Initializing\n"));

    netsnmp_register_scalar(netsnmp_create_handler_registration
                        ("volume", handle_volume, volume_oid,
                        OID_LENGTH(volume_oid), HANDLER_CAN_RWRITE));
}

int
handle_volume(netsnmp_mib_handler *handler,
              netsnmp_handler_registration *reginfo,
              netsnmp_agent_request_info *reqinfo,
              netsnmp_request_info *requests)
{
    int ret;
    /*
     * We are never called for a GETNEXT if it's registered as a
     * "instance", as it's "magically" handled for us.
     */

    /*
     * a instance handler also only hands us one request at a time, so
     * we don't need to loop over a list of requests; we'll only get one.
     */

    switch (reqinfo->mode) {
```
case MODE_GET:
    snmp_set_var_typed_value(requests->requestvb, ASN_INTEGER,
                      (u_char *)
                      /* XXX: a pointer to the scalar's data */
                      ,
                      /*
                      * XXX: the length of the data in bytes
                      */
                  );
    break;

    /*
    * SET REQUEST
    *
    * multiple states in the transaction. See:
    */
    case MODE_SET_RESERVE1:
        /*
        * or you could use netsnmp_check_vb_type_and_size instead
        */
        ret = netsnmp_check_vb_type(requests->requestvb, ASN_INTEGER);
        if (ret != SNMP_ERR_NOERROR) {
            netsnmp_set_request_error(reqinfo, requests, ret);
        }
        break;

    case MODE_SET_RESERVE2:
        /*
        * XXX malloc "undo" storage buffer
        */
        if ( /* XXX if malloc, or whatever, failed: */ ) {
            netsnmp_set_request_error(reqinfo, requests, SNMP_ERR_RESOURCEUNAVAILABLE);
        }
        break;

    case MODE_SET_FREE:
        /*
        * XXX: free resources allocated in RESERVE1 and/or
        * RESERVE2. Something failed somewhere, and the states
        * below won't be called.
        */
        break;

    case MODE_SET_ACTION:

86
/*
 * XXX: perform the value change here
 */
if ( /* XXX: error? */ ) {
    netsnmp_set_request_error(reqinfo, requests, /* some error */);
}
break;

case MODE_SET_COMMIT:
    /*
     * XXX: delete temporary storage
     */
if ( /* XXX: error? */ ) {
    /*
     * try _really_really_ hard to never get to this point
     */
    netsnmp_set_request_error(reqinfo, requests,
        SNMP_ERR_COMMITFAILED);
}
break;

case MODE_SET_UNDO:
    /*
     * XXX: UNDO and return to previous value for the object
     */
if ( /* XXX: error? */ ) {
    /*
     * try _really_really_ hard to never get to this point
     */
    netsnmp_set_request_error(reqinfo, requests,
        SNMP_ERR_UNDOFAILED);
}
break;

default:
    /*
     * we should never get here, so this is a really bad error
     */
    snmp_log(LOG_ERR, "unknown mode (%d) in handle_volume\n", reqinfo->mode);
    return SNMP_ERR_GENERR;
}
return SNMP_ERR_NOERROR;