Großer Beleg

A Sound Server for DROPS

Christoph Müller

2nd March 2006

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter:	Dipl.-Inf. Christian Helmuth

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 2. März 2006

Christoph Müller

Contents

1 Introduction **1**

2 State Of The Art **3**

2.1 Operating Systems . . . 3
2.1.1 Linux . . . 3
2.1.2 Mac OS X . . . 3
2.1.3 Windows . . . 4
2.2 Drivers . . . 4
2.2.1 OSS . . . 4
2.2.2 ALSA . . . 5
2.3 Sound Servers . . . 5
2.3.1 aRtsd . . . 6
2.3.2 ESD . . . 6
2.3.3 JACK . . . 6
2.4 Plugin architectures . . . 6
2.4.1 VST . . . 6
2.4.2 LADSPA . . . 6
2.4.3 JACK . . . 7
2.5 Already existing solutions for DROPS . . . 7
2.5.1 OSS library . . . 7
2.5.2 DDE26 . . . 7
2.5.3 dsound . . . 8
2.5.4 DSI . . . 8

3 Design **9**

3.1 Objectives . . . 9
3.2 Overview . . . 9
3.3 Interfaces . . . 12
3.3.1 Rack . . . 14
3.3.2 Component . . . 14
3.4 Libraries . . . 15
3.4.1 Rack . . . 15
3.4.2 Component . . . 16
3.5 ALSA Sink . . . 16
3.6 More Sinks . . . 17

3.7 Mixing Server 18
3.8 Player 18

4 Implementation **19**
4.1 Component Library 19
4.2 ALSA Sink 19
4.2.1 alsa-kernel-core 20
4.2.2 alsa-kernel 21
4.2.3 alsa-lib 21
4.2.4 Server 21
4.3 Mixer 22
4.4 Source 22

5 Evaluation **23**

6 Conclusion And Outlook **25**
6.1 Future Work 25

Glossary **27**

Bibliography **29**

List of Figures

3.1 An examplary scenario. 10
3.2 An unordered schedule. 10
3.3 An ordered schedule. 10
3.4 An Example Schedule 13

4.1 Overview of the alsa server 20

List of Tables

3.1 Total space required for buffers against number of interlinked applications with equal total latency. 11
3.2 Attributes of a port. 12
3.3 Attributes of a property. 13

4.1 Number of frames per period, depending on sampling rate and frame size. 19

1 Introduction

The DROPS project [dro] is a research operating system from the Technical University of Dresden. It is based on the L4/Fiasco microkernel [Lie96][fia], which is also being developed at TU Dresden. Its goal is to run real-time [1] and non real-time applications side by side. Because real-time applications reserve resources for the worst case and rarely use the complete reservations, these unused reservations can be used by non real-time applications, which results in a higher utilization of the resources, rather than using only real-time applications.

This work focuses on developing an audio subsystem for DROPS. It will support real-time and non real-time applications, and provide a mechanism for flexible interconnection of applications at runtime. Thus real-time applications like a video-conference-system get the requested resources to do their work and non real-time applications like system beeps will be played when the system can afford it. By using a software mixer it will be possible to use one sound device for several applications and the Linux ALSA drivers have been ported to provide support for a large amount of devices.

The next chapter gives the reader an overview of existing audio architectures. Following this, the design will be explained. The fourth chapter is about implementation details and the subsequent one will evaluate design and implementation. The last section lists issues to be addressed and provides suggestions for future work.

[1] A real-time application requests its amount of resources and if the system can afford it, it guarantees that the application under all circumstances gets its reservations.

2 State Of The Art

This section is intended to give the reader an overview of existing audio architectures.

2.1 Operating Systems

2.1.1 Linux

A common Linux installation is not optimized for audio streams. Therefore it happens that playback is interrupted (e.g. by repaint operations from the X window system).

There are two approaches towards uninterrupted sound processing in Linux.

One reduces scheduling latency by making kernel routines preemptible. This increases responsiveness and improves the situation, but introduces neither any guaranties nor real-time scheduling.

The other was developed by ARVID STRAUB [str04]. It provides real-time audio processing on Linux. This is achieved by using existing real-time patches, which introduce a real-time scheduler that is superior to the common Linux scheduler. Because of a real-time interface for sound drivers, this solution requires special modified drivers for the sound hardware. All real-time processing of audio data is done in kernel mode. A plugin API declares an interface for audio processing kernel modules and special module called router does the central management.

2.1.2 Mac OS X

The latest operating system from Apple is based on a Mach microkernel and a BSD kernel [App]. It has support for real-time threads[1].

[1] A real-time thread must define how many cycles it requires to compute.

The architecture for audio drivers is documented in [App06]. Between audio drivers and applications there is a hardware abstraction layer (HAL). Applications and HAL use only 32 bit floating point data. It is the driver's duty to convert the samples to a format understood by the hardware.

The driver also takes timestamps when the DMA engine wraps to the beginning of the ringbuffer. These timestamps are read by the HAL which estimates the point in time when clients have to deliver the next portion of data. At these moments it activates the i/o threads of the applications and the applications add their samples to a common buffer.

2.1.3 Windows

In modern variants of the operating system from Microsoft (i.e. 2000 and XP), there is a technique called kernel streaming [Mic03]. It does processing of streamed data in kernel mode.

2.2 Drivers

For being able to support a wide range of hardware, it could be useful to reuse an existing set of drivers for sound hardware.

Because of legal issues and lack of source code, it is not possible to use sound drivers from Windows or Mac OS.

There are two distinct sets of drivers for Linux: OSS and ALSA. Both of them support a lot of devices. Their source code is part of the linux kernel, which makes it possible to reuse them legally.

2.2.1 OSS

The open sound system is a product of 4Front Technologies [4Fr99]. It provides sound drivers for a couple of operating systems, e.g. Solaris, Linux and BSD variants.

A subset of this product is available as OSS/Free and has been part of the linux kernel since version 2.0.3x. The kernel interface for OSS drivers is relatively simple. An application sets the parameters for the stream via `ioctl` and sends the samples to the driver using an ordinary `write` call.

2.2.2 ALSA

With the release of version 2.6, the advanced linux sound architecture [als] has superseded the OSS project as sound driver architecture for Linux. It is a community development, supported by the SuSE company and released unter the GPL. ALSA consists of kernel modules and a library.

The kernel modules contain the core functionalities and drivers for each device. Because the core provides everything a driver needs, only symbols from the core are referenced in the driver modules.

As the kernel interface is based on `ioctl` calls having a pointer to a structure in memory as parameter, it is not easy to use it directly. This is no problem because the ALSA library does this and provides a simpler interface. The library provides plugins for various purposes.

There are plugin modules for sampling rate conversion, format conversion, channel routing and more. These plugins are configured in the ALSA configuration file, which consists of sections defining virtual devices and a chain of plugins behind these devices. An application connects to these virtual devices using calls from the library.

The dmix plugin makes it possible that more than one application may play sound at the same time. When the first application starts using dmix, a separate process is forked. This is the dmix server. Each application adds the samples of its stream to a common shared buffer and the mixing server sends the sum to the driver. Synchronisation is done using sockets.

By providing an OSS emulation, ALSA enables OSS applications to run without any modification.

2.3 Sound Servers

Because OSS is not able to support more than one application at the same time, sound servers have been developed. This is a brief overview of the most important implementations.

2.3.1 aRtsd

aRtsd is the sound mixer of KDE [kde], a session manager for Linux. Applications that want to utilize it, have to use a special library for sound output. It also provides plugins for miscellaneous purposes.

2.3.2 ESD

ESD is the counterpart for Gnome [gno], also a session manager. It does only mixing of applications onto one sound device.

2.3.3 JACK

Because the existing sound servers stream data only from applications to the driver, JACK was developed [jac]. It does also mixing of several applications onto one device, but in addition it is able to stream data between applications.

2.4 Plugin architectures

2.4.1 VST

The virtual studio technology was developed by Steinberg Media Technologies GmbH [ste]. It is an interface between an application (host) and dynamic linked libraries (plugins). All samples are 32 bit floating point numbers, optional 64 bit. Each plugin provides a callback function that has three parameters: a pointer to the source buffer, a pointer do the destination buffer and a number indicating the amount of data to be processed.

2.4.2 LADSPA

Because there have been legal issues using VST on linux, LADSPA, the linux audio developer's simple plugin api, has been developed [lad]. It also distinguishes between a host and plugins and all samples are 32 bit floating point numbers.

A shared library may contain more than one plugin. Each plugin has control and data ports. Control ports can be used to configure properties of the plugin (e.g. the delay for an echo plugin). Data ports are connected using `connect_port`. This assings a memory location to the port, which will be used as shared buffer between host and plugin.

There are two callbacks for processing data: `run` and `run_adding`. Both take as parameters a pointer to a plugin instance and a number of samples to process. The first overwrites any data at the destination buffer, the latter one adds the result to the destination buffer.

2.4.3 JACK

As stated before, JACK provides inter-application streaming (external clients). It also supports clients that can be plugged into the JACK server (internal clients). Shared memory is used for transferring the samples between the address spaces.

Each client provides a `process` callback. For internal clients, this is invoked periodically by the server. For external clients, each client's process call is activated by the previous client (the one it receives the samples from).

Streams consist of 32 bit floating point data and only one channel. For stereo, two streams have to be established.

2.5 Already existing solutions for DROPS

2.5.1 OSS library

CHRISTIAN HELMUTH [Hel01] developed a library that uses an emulation layer and sound drivers from Linux version 2.4. Using this library, an application is able to play sound. There exists a variant of this library for each supported device.

2.5.2 DDE26

As a continuation of the work of HELMUT, MAREK MENZER [Men04] developed an emulation layer, easing porting of drivers from Linux version 2.6. It is based on Linux version 2.6.6.

2.5.3 dsound

While porting the SDL library [lib] to DROPS, THOMAS FRIEBEL [Fri05] wrote a sound server that utilizes the OSS library. This implementation provides playback for one SDL application.

2.5.4 DSI

The drops streaming interface [LRH01] provides a library for streaming data from one task to another. It uses shared memory and divides the stream into packets which are sequentially written to the shared buffer. The buffer may contain more than one packets. All required memory is allocated before begin of transfer.

The producer is only blocked if the buffer is full and the receiver only if the buffer is empty. So, if consumer and producer are scheduled in an effective way, the buffer is neverfull nor empty and thus no blocking occurs. There is also a mode that indicates an error instead of blocking.

3 Design

3.1 Objectives

The main objective of this work is developing a solution, that makes it possible to mix the output of multiple programs onto one sound device. Routing n sources to m sinks will also be possible, but is not deeply discussed in this work. Sound may be any type of pulse-code modulated time discrete sampled signal. Because of their different nature, other formats like MIDI are not considered.

For compatibility reasons the design should support a wide range of audio hardware. It also needs to be secure, meaning that no program must have access to another program's data unless it is intended. In addition misbehaving programs must be taken into consideration. Their influence on other applications must be minimized. For time-critical applications it is important to have low latency and for quality reasons loss of samples has to be avoided. Naturally, it should efficiently utilize resources and last, a mechanism for flexible configuration at runtime improves usability.

The next section introduces the basic topology of the design. Then the interfaces are introduced, followed by the concepts for the libraries implementing the interfaces. The last part of this chapter is about sample implementations.

3.2 Overview

The first idea was to build one server by porting ALSA (section 2.2.2) to L4. ALSA was chosen, because it is currently the default sound architecture for Linux and thus future devices may not be supported by OSS (section 2.2.1).

In this approach, all clients deliver their data at the server and an instance of the dmix-plugin adds up the streams. The benefit from this approach is, that a huge amount of code from the ALSA project can be reused. On the other hand, especially the dmix plugin requires a lot of posix functionality (e.g. sockets, shared memory, fork), making the portation much more complex.

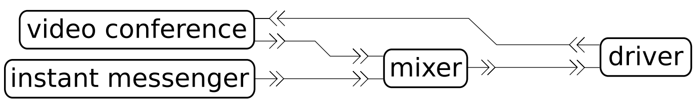

Figure 3.1: An examplary scenario.

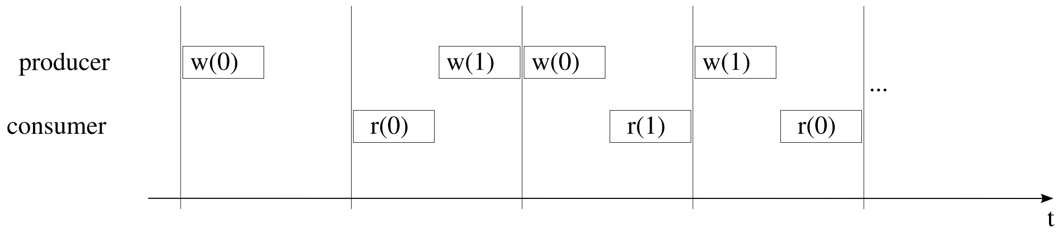

Figure 3.2: An unordered schedule.

This first approach was superseded by the second one, which is more flexible. It introduces a common interface for audio streams. Every audio processing application has input or output ports or both of them. These connection points may be connected at runtime. A mixing server has to be used, if mixing of several streams into one is needed. Several mixing servers can be implemented, each one optimized for a specific situation and it is possible to have a mixer implementation with more than one output, providing the ability to route *n* applications to *m* sinks.

Having a common audio interface, servers for various purposes can be interlinked, e.g. sampling rate conversion, effect processing or network streaming. Figure 3.1 shows a possible scenario.

The benefit from the first approach can still be used by implementing an ALSA server. If the hardware is able to mix several streams, the ALSA server may provide several inputs and the separate mixing server can be omitted.

Because streaming audio data is a continuous job, each audio application has a worker thread that processes periodically portions of the stream.

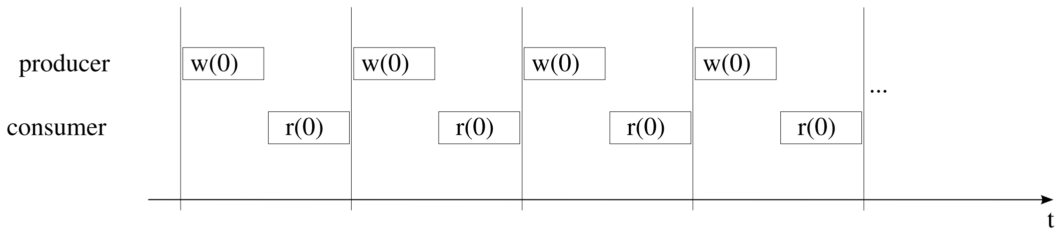

Figure 3.3: An ordered schedule.

n	unordered	ordered
2	$f(T_{\text{ordered}})$	$f(T_{\text{ordered}})$
3	$\frac{4}{3}f(T_{\text{ordered}})$	$f2 \cdot (T_{\text{ordered}})$
4	$\frac{6}{4}f(T_{\text{ordered}})$	$f3 \cdot (T_{\text{ordered}})$
5	$\frac{8}{5}f(T_{\text{ordered}})$	$f4 \cdot (T_{\text{ordered}})$
∞	$2 \cdot f(T_{\text{ordered}})$	∞

Table 3.1: Total space required for buffers against number of interlinked applications with equal total latency.

Figure 3.2 shows an example schedule for a single stream between two applications. Because there may be more than two servers interlinked, the number of servers is reffered to as n. Each of these n servers processes one portion of data per period of size T. The latency l is the time, a sample needs to pass all interlinked servers. $f(t)$ denotes the number of frames[1], that have to be processed within a period of size t. There are two aproaches.

- Unordered approach: In period p each server processes the data, which was produced by its predecessor in period $p-1$. This is illustrated in figure 3.2. The latency depends on the number of interlinked applications ($l = n \cdot T_{\text{unordered}}$). It requires two buffers per connection, which are alternately read and written. Therefore the required space for buffers b can be calculated as

$$b = 2 \cdot f(T_{\text{unordered}}) \cdot (n-1)$$

- Ordered approach: A portion of data traverses all applications within one period. This is diagrammed in figure 3.3. The latency is constant and equal to the size of a period ($l = T_{\text{ordered}}$). There are data dependencies between the applications, complicating scheduling. Only one buffer is required for each connection. The sum of all buffers is $f(T_{\text{ordered}}) \cdot (n-1)$.

It seems that the ordered approach results in a smaller latency, but the data dependencies that appear make scheduling very complex. The same latency can be achieved by using the unordered approach with a period size of $T_{unordered} = \frac{T_{\text{ordered}}}{n}$. The buffer size b is then

$$b = 2 \cdot f(\frac{T_{\text{ordered}}}{n}) \cdot (n-1)$$

Because the number of frames per period is proportional to the period size, it can be written as

$$\frac{2 \cdot f(T_{\text{ordered}}) \cdot (n-1)}{n}$$

Table 3.1 lists the amount of buffer space required for both approaches. It shows,

[1] A frame consists of one sample for each channel.

attribute name	description
name	A human readable string describing this port
type	It determines whether this is a source or a sink.
format	The type of the frames. For uncompressed streams this indicates byte order and sample type, e.g. 16 bit signed little endian. In case of compressed audio stream this may denote the algorithm and its parameters.
rate	The sampling rate, i.e. number of frames per second.
channels	The number of channels. e.g. 2 for a stereo stream.

Table 3.2: Attributes of a port.

that the unordered approach with smaller period size and the same latency requires less memory for buffers if more than two applications are interlinked. Differently explained: using the unordered approach does not increase the buffer space (if the period size is $\frac{T_{\text{ordered}}}{n}$), although it uses double buffering.

Because scheduling with data dependencies is very complex and the ordered approach does not have any significant advantages, the unordered approach was chosen. The only drawback is, that the latency now depends on the number of interlinked servers, but this is also in the domain of physical devices and therefore acceptable.

3.3 Interfaces

As stated in the previous section, an interface for streaming audio data from one task to another is needed. The first task will be named `source` and the second one `sink`. Both are `components` of the audio system.

There exists already an implementation for streaming data: DSI (section 2.5.4). Because it provides mechanisms for shared memory and synchronisation, it will be used for transferring data between tasks[2].

Besides source and sink, DSI requires a management application. The management functionality could be integrated into one of the components, but this leads to unwanted inflexibility, because in every setup this component has to be used. Building a separate management server is a better approach. It will be called a `rack`.

Inputs and outputs of components are ports. Table 3.2 lists the attributes of a port.

Some components may have configurable properties, e.g. the mixing server has gain for each input. There may also be properties not belonging to a port, e.g. parameters

[2] A task is an address space where threads reside.

attribute name	description
name	A human readable string describing this property.
type	It determines whether this is a boolean switch or a numeric value. For numeric properties it specifies also a range of valid values.

Table 3.3: Attributes of a property.

for an effect processor module. Hence properties belong either to a component or to a port of a component. Table 3.3 shows the attributes of a property.

Components register at the rack, thus they tell the rack about their ports and properties. The rack opens and connects ports using DSI calls. Only ports with identical properties and different directions can be connected. This keeps connection handling simple.

Converting servers may be implemented. They could have different inputs, one for each supported type or a converting server is started on demand with only the required ports.

Components can be real-time components or best-effort (non real-time) components.

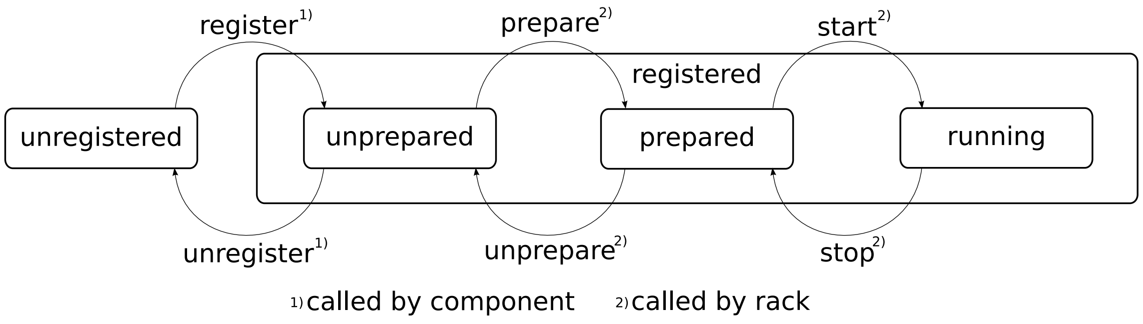

Figure 3.4: An Example Schedule

Figure 3.4 illustrates the states and transitions for components. After registering, a component is in the unprepared state.

By using the `prepare` call, the rack triggers preparation of the component. This creates the worker thread and one thread for receiving preemption IPCs. Real-time components have to reserve the required amount of processing time at the admission server. Best-effort components do not reserve processing time. They set the parameters for the periodic execution of the worker thread using `rt_add_timeslice`[3] with deadline and worst case execution time equal to the period size. This way preemption and timeslice overrun IPCs only occur if the component was unable to complete processing for a period, which can be used to detect it, display an error message and reset the

[3] For details about periodic scheduling, please consult UDO STEINBERG's thesis [Ste04].

worker thread. To assure that real-time components get their reserved time, they must have a higher priority than the non real-time components.

The `start` call activates a component. This invokes `rt_begin_periodic`, which triggers the begin of the periodic execution. The `stop` call causes `rt_end_periodic`, which will halt the worker thread. To free up reserved time and reset the component to the unprepared state, `unprepare` must be used.

To make sure that the periods of all components are synchronous, they start when the time modulo period size is zero.

The rack may connect and disconnect ports on running, prepared and unprepared components. So the resulting system can be flexibly configured at runtime.

Since there are requests from both sides, e.g. a component registers at the rack and the rack opens a port, the interface is splitted into two. The first one contains all calls served by the rack, the second one all for the reverse direction.

3.3.1 Rack

Components are clients to the rack server. They register at the rack using `component_-register`. It requires a structure describing this component, its ports and properties as parameter. This structure also declares the thread listening for requests and whether or not this is a real-time component.

Because different racks may use different period sizes and a component has to know the size of the period prior to registering, there is a `get_period_size` call.

An `unregister` call completes the rack interface. Only unprepared components with no open ports may be unregistered.

3.3.2 Component

The rack is client for the component servers. There are methods for opening and connecting ports (`[source|sink]_open`, `ports_connect`).

The dataspace for the shared buffer is allocated by the source, because this may allow optimisations, e.g. the dataspace is identical to the dataspace where the file resides that is played. Therefore the open calls slightly differ. A `source_open` gets the port number and returns a socket reference and references to control dataspace and data dataspace. On a sink the dataspaces are input values. `prepare`, `start`, `stop` and `unprepare` have already been introduced above.

Properties are read using `property_read` and set with `property_write`. Although this is sufficient for most cases, components may implement their own additional configuration interfaces. It is also possible, that a component is an application having its own graphical user interface.

3.4 Libraries

To ease the implementation of components, there are libraries wrapping the details of the communication (i.e. DSI) and providing a simple interface for audio applications.

3.4.1 Rack

The rack has to manage the components. Its library contains all DSI-calls and structures and hides them from the implementation. This keeps the interface free from DSI types. Components, ports and properties are referred to by using natural numbers and the library resolves these numbers to the corresponding data structures. Because the period size is not fixed and may be chosen by a rack implementation, there is `set_period_size` procedure.

The library provides a call for creating the server thread which will be listening for register requests. On an incoming event, the appropriate callback function is invoked to inform the implementation about this event (register or unregister).

The component describing structure is visible to the implementation. It contains definitions of ports and properties and the information whether this is a real-time component.

Using `ports_compatible` it is possible to determine, if a pair of ports can be connected, which is then done with `ports_connect`. This automatically opens the requested ports and connects them.

The library also provides calls that are forwarded to the component. These are `prepare`, `start`, `stop`, `unprepare`, `property_read` and `property_write`.

3.4.2 Component

The component library manages registering at rack, handling sockets and property requests. It requests the period size from the rack and calculates the buffer size.

Because the id of the serving thread is part of the component description (so that the rack knows, where to ask), there is no `component_register` call. This is done during startup of the server loop. Before it, the implementation has to construct a structure describing its ports and properties.

There is a callback (`allocate_ds`), which is called if a source port is being opened. It allocates the dataspace used as shared data buffer. Allocating the shared area is done by the source, because this can be used for an optimization: If the source plays sound from files, it may use the file's dataspace for the shared area. In this way the source must not copy the samples from file to buffer.

There are callback functions that handle property read and write requests. If a deadline was missed or a timeslice overrun occured, the implementation is informed via another callback.

3.5 ALSA Sink

This explains the design of the audio sink based on the Linux ALSA drivers.

ALSA was chosen, because it has been the default audio driver architecture since linux 2.6. It supports most of the hardware found in modern computers and once the core is running, all drivers should operate and there is no need to change anything in the drivers.

To ease migration to a newer version of ALSA, changes at the ALSA-code must be minimized. Therefore the device driver environment for linux 2.6 must be used (section 2.5.2).

Because the ALSA kernel interface is based on `ioctl`-calls having pointers to datastructures as parameters, it is very difficult to use it directly. The ALSA library wraps them and provides a more usable interface.

There are three different ways how to use Alsa library and kernel modules:

- The ALSA library is linked to the client (i.e. a source component). This is not possible without bloating the previous declared interface.

- The library code resides in a separate server, that acts as a sink and has a special interface to the server containing the ALSA drivers. The advantage of this approach is that there is a clean task barrier between kernel drivers and ALSA library.

- The problem of `ioctl` calls with pointers as parameters leads to a third solution: Both, library code and kernel drivers, are situated in one server that acts as a sink. Because both share the same address space, it is now easy to implement these ioctl calls.

By implementing the second approach, one would have to use a large area of shared memory where all allocated structures that may be used by `ioctl` are located. Because this increases complexity and decreases protection by the task boundary, which was its advantage over the third approach, the third can be used.

This paragraph explains how the ALSA sink works. On startup all required DDE modules are initialized, then the core modules of the ALSA drivers followed by the drivers. Multiple drivers may be compiled into the server. Only those finding a supported device will become active. After that point all kernel level services are ready. Now basic mixer settings have to be applied, depending on the card that was detected. Then the component library gets initialized and the required threads started. The server provides one or more sink ports, depending on the number of devices. These ports should provide the highest quality, that is needed by the system and supported by the device. If a stream with different parameters must be connected, a converting server has to be used.

The incoming buffer is periodically read by the worker thread and the frames are sent to the driver using the write function of the alsa library.

There are soundcards that are capable of mixing several streams in hardware and ALSA supports it. If such a device is present, it should be possible to provide several input ports that are mixed by the hardware.

For recording, the ALSA server provides one or more source ports.

3.6 More Sinks

It is possible to implement other audio sinks. There could be an implementation using the OSS drivers from linux 2.4. For fiasco-ux an implementation that uses the drivers of the host system is possible.

3.7 Mixing Server

This section explains the design of a mixing server.

It will have several input ports and one output port. Each input port has a property defining the gain for it. All ports have the same type. Format conversion is not done by this mixing server. Separate converters must be used.

All incoming samples are added up and if the range limit for a sample is exceeded, the sample's value will be set to the maximum or minimum value.

The processing thread reads all inputs and adds them to the output buffer. For the first input, it copies instead of adding, so zeroing-out the output buffer is not necessary.

3.8 Player

A simple player for pcm data gets its input from a file server or as module loaded at boot time. In both cases the input data is an array in memory. The worker thread copies the required amount of frames to the shared buffer.

4 Implementation

This chapter tells the reader details about the implementation.

By sending a stream of samples chopped into periodically processed packets, there are combinations of period sizes and sampling rates that do not consort. Table 4.1 illustrates this by listing frame counts for common sampling rates.

All existing implementations are non real-time components, due to the lack of time for investigations about determining their worst-case execution times. The next sections will discuss the different libraries and servers.

4.1 Component Library

While implementing the sound components, it bekame known that the worker thread still had to use DSI (section 2.5.4) calls for getting and committing packets. Therefore `[sink|source]_packet_get` and `[sink|source]_packet_commit` were added to the library. They wrap the DSI calls and have only the port number and a pointer to the packet's data as parameter.

4.2 ALSA Sink

The package `l4alsa` contains the ported ALSA parts (section 2.2.2) and the ALSA server. It provides one input port for playback of a 16 bit signed little endian stereo

rate in kHz / period in ms	8	11.025	16	22.05	44.1	48
100	800	1102.5	1600	2205	4410	4800
10	80	110.25	160	220.5	441	480
1	8	11.025	16	22.05	44.1	48

Table 4.1: Number of frames per period, depending on sampling rate and frame size.

stream at 44.1 kHz. Though capture was planned, it has not yet been implemented. The server is based on ALSA version 1.0.4rc2, because the DDE26-library (section 2.5.2) implements the kernel interface from Linux 2.6.6, which was released with this ALSA version[1].

The ALSA drivers can be found in the `alsa-kernel-core` library, the ALSA library is located in the library named `alsa-lib` and `alsa-kernel` provides an interface to the drivers. Figure 4.1 illustrates the interaction of the modules.

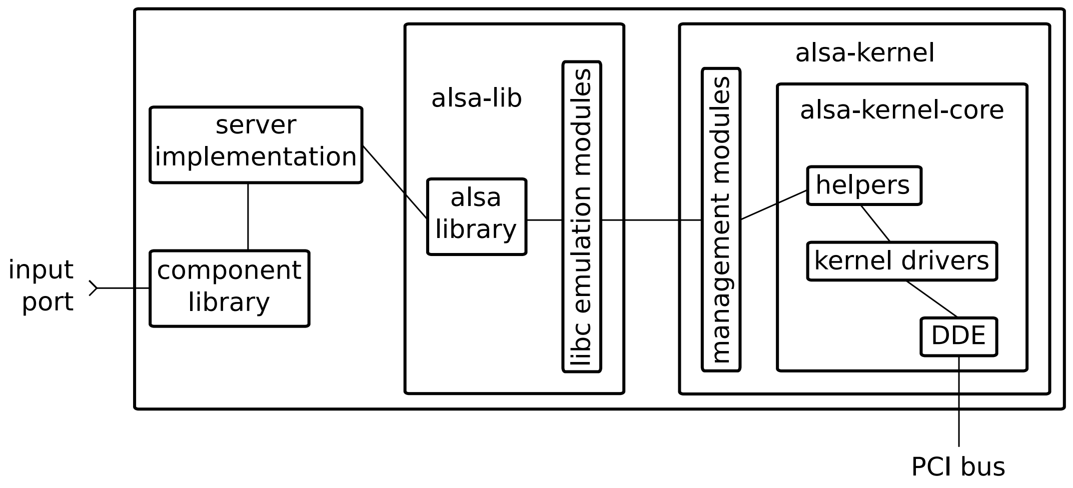

Figure 4.1: Overview of the alsa server

4.2.1 alsa-kernel-core

This contains the drivers. It also implements additional Linux services not provided by DDE (i.e. `fget`, `fput`, `vscnprintf`). Instead of implementing these calls, one could have changed the corresponding places in the ALSA code. But this increases the changes made, which complicates upgrades to newer versions of ALSA. Furthermore it implements `do_gettimeofday` by using `gettimeofday` from `libc_be_time` as time source.

The drivers of three devices have been ported, but only the one for `es1938` (aka. solo1) did function. Due to problems with PCI access, the `ens1371` driver refused to work. There may be an issue within DDE. Third, the `au88x0` driver did not cause an error, but this device contains a routing chip that has to be configured. The right configuration was not found and thus sound never reached the phone jack.

[1] see linux/include/sound/version.h

4.2.2 alsa-kernel

This provides an interface to the kernel drivers. It implements `devfs_mk_cdev`, which is called by each driver during initialisation, and uses it for getting information about existing devices. The list of devices contains `inode` and `file` structures which are used by the `ioctl` implementation.

During build, `alsa-kernel-code`, `dde_linux26` and `alsa-kernel` are linked together, keeping only interface methods global. This prevents namespace clashes between ALSA library and drivers.

4.2.3 alsa-lib

This includes the ALSA library and glue code that implements required functionality from the C library or redirects requests to the `alsa-kernel` library.

The ALSA library requires a configuration file that defines plugins and virtual devices. The most simple way to implement it is linking the file as character array to the library. This does not require an external file server and because the sever uses the library in a predefined constant way, it is no drawback to have it defined at compile time. The function `snd_input_stdio_open` is used to access files. There is another function taking a pointer to a buffer instead of a filename (`snd_input_buffer_open`). The first one was modified to redirect the request to the second one.

The library uses `mmap` to get references to driver data structures. Because driver and library reside in the same task, a real mapping operation is not necessary. Therefore `alsa-kernel` provides a `mmap` call that determines the address and returns it.

4.2.4 Server

The server implementation utilizes the above mentioned libraries in a way shown in figure 4.1. Because every sound device features a mixer that has to be configured, the server does this using the ALSA library's interface. The code of the `amixer` program[2] is used for this purpose. Currently, the mixer setup routine is built for the `es1938` device only.

Data processing is done by two threads. The worker thread is the one being periodically invoked by the kernel. It gets a packet of data and copies it to a ring buffer. The writer is the second one. It reads data from the buffer and sends it to the driver

[2] found in the alsa-utilities package

using `snd_pcm_writei` from the ALSA library. This setup is necessary, because the write call blocks until all frames have been copied to the driver's buffer, which is not compatible with being periodically scheduled.

4.3 Mixer

The mixer is the implementation of the design in secton 3.7. It delivers the sum of its two inputs as output. An extension to more than two inputs should be possible without much effort, but implementing adjustable gain for the ports is more difficult and therefore not yet done.

4.4 Source

This is the only source existing at the moment. It sends an infinite stream by looping over a buffer of samples, which is loaded at boot time. The only problem with it was that debugging output inside the processing loop caused a delay, which resulted in deadline misses.

5 Evaluation

The implementation has been tested on a Pentium II based system running at 233 MHz. Two Sources playing different sounds were added by the mixer and then played via the ALSA server.

The number of copies is a gage for efficiency because streaming of digital audio is based on copying the samples from one buffer to another

Each producing component has to write the processed samples into the shared buffer, whereas the consuming component reads them. This results in one copy operation for each interlinked component. For source, mixer and driver this makes three. Additionally, the ALSA server has an internal ringbuffer, incrementing the number of copy operations by one.

For comparison, ALSA on Linux requires only two copy operations. First, an application calls `write`, which adds the samples to the shared buffer provided by dmix. Then dmix copies the data to the driver buffer, which is read by the device.

If the ALSA server is modified to work without this extra buffer (e.g. by using non-blocking write) there is still one copy over the two used in Linux. This is unavoidable as long as the constraint that no client may see another client's data is valid, because a dmix client is able to read the data from the shared buffer, which may already contain samples from other applications. This is not possible in this works design, due to the additional copying.

Various sampling rate and period size combinations have been tested. Sometimes the ALSA driver reports a buffer underrun. This could be caused if the writer thread returns from a write call and is preempted before it can issue the next write.

If the period is shorter than 100 ms, displaying status messages causes so much delay, that deadlines are missed. After removing these outputs, it worked well with 44.1 kHz and a period size of 10 ms.

6 Conclusion And Outlook

This work presents an audio architecture for microkernel based systems, especially DROPS [dro]. It provides an interface for streaming data. Applications that implement this interface can be flexibly interconnected. Exemplary applications have been developed, including a simple player, a software mixer and a driver server. For the driver server, ALSA (section 2.2.2) has been ported.

6.1 Future Work

The current ALSA server implementation could be improved. It has to be investigated, whether or not a non blocking `snd_pcm_writei` can be used to eliminate the buffer underruns in the driver and if this eliminates the need for the intermediate buffer. More drivers should be ported and tested and the mixer setup routine has to be rewritten to use the right setup for each device.

There are more components possible: a driver server using OSS drivers (section 2.2.1), an audio sink for fiasco-ux[1] utilizing the host sytem's drivers, converters for sampling and sample format, a sink and source pair for network streaming and countless effect processing servers. An output module for verner [Rie03] completes the list of possible components.

The existing components could be modified to become real-time components or new real-time components could be developed.

[1] A version of the fiasco microkernel running on Linux in user space.

Glossary

ALSA Advanced Linux Sound Architecture

API Application Programming Interface

aRts Analog Real-Time Synthesizer, sound system of KDE

BSD Berkeley Software Distribution, a variant of the unix operating system

DDE Device Driver Environment, a library providing and emulating linux services for drivers

DMA Direct Memory Access, a techique where a device directly accesses main memory

DROPS Dresden Real-Time Operating System

DSI Drops Streaming Interface

ESD the Enlightened Sound Daemon, sound mixer of GNOME

GNOME GNU Object Model Environment

GPL GNU General Public License

HAL Hardware Abstraction Layer, library that simplifies hardware access

IPC Inter Process Communication

JACK JACK Audio Connection Kit

KDE K Desktop Environment

LADSPA Linux Audio Developer's Simple Plugin API

MIDI Musical Instrument Digital Interface

OSS	Open Sound System
PCI	Peripheral Component Interconnect standard, a bus system for computers
SDL	Simple DirectMedia Layer
VST	Virtual Studio Technology

Bibliography

[4Fr99] 4Front Technologies. 4Front Technologies. http://www.4front-tech.com/ossfree/, January 1999. 4

[als] Alsa project website. http://www.alsa-project.org. 5

[App] Apple Computer, Inc. Darwin documentation. http://developer.apple.com/darwin/. 3

[App06] Apple Computer, Inc. Audio device driver programming guide. http://developer.apple.com/documentation/DeviceDrivers/Conceptual/WritingAudioDrivers/index.html, January 2006. 4

[dro] Dresden real-time operating system website. http://os.inf.tu-dresden.de/drops/. 1, 25

[fia] Fiasco microkernel website. http://os.inf.tu-dresden.de/fiasco/. 1

[Fri05] Thomas Friebel. Portierung der libsdl auf drops / dope. http://os.inf.tu-dresden.de/papers_ps/friebel-beleg.pdf, May 2005. 8

[gno] GNOME: The free software desktop project. http://www.gnome.org. 6

[Hel01] Christian Helmuth. Generische Portierung von Linux-Gerätetreibern auf die DROPS-Architektur. Master's thesis, TU Dresden, 2001. Available from: `http://os.inf.tu-dresden.de/project/finished/finished.xml.de#helmuth-diplom`. 7

[jac] Jack audio connection kit. http://jackit.sourceforge.net/. 6

[kde] K desktop environment. http://www.kde.org. 6

[lad] Linux Audio Developer's Simple Plugin API. www.ladspa.org. 6

[lib] Simple directmedia layer. http://www.libsdl.org. 8

[Lie96] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996. 1

[LRH01] Jork Löser, Lars Reuther, and Hermann Härtig. A streaming interface for real-time interprocess communication. Technical Report TUD-FI01-09-August-2001, TU Dresden, August 2001. Available from URL: `http://os.inf.tu-dresden.de/~jork/dsi_tech_200108.ps`. 8

[Men04] Marek Menzer. Portierung des drops device driver environment (dde) fÃ¼r linux 2.6 am beispiel des ide-treibers, January 2004. 7

[Mic03] Microsoft Corporation. Windows kernel streaming architecture. http://www.microsoft.com/whdc/archive/csa1.mspx, April 2003. 4

[Rie03] Carsten Rietzschel. VERNER – ein Video EnkodeR uNd playER für DROPS, Oktober 2003. 25

[ste] Steinberg gmbh. http://www.steinberg.de. 6

[Ste04] Udo Steinberg. Quality-Assuring Scheduling in the Fiasco Microkernel. Master's thesis, TU Dresden, March 2004. 13

[str04] RAT - Real-Time Audio Tools. Technical report, April 2004. 3