Ontwerpproject 2002, RiVo

Matthijs van der Kooij, Barry Nijenhuis, Jeroen Soesbergen, Ardjan Zwartjes
supervisors: Pierre Jansen, Hans Scholten

8th March 2002

Chapter 1

Summary

Rivo is a digital radio recorder and player, supporting time-shifting. We built this as part of the Designproject 2002. It has been split up into 4 main modules: audio, planner, streamer and interface. We divided the responsibilities over the participants. This separation of concerns worked out nicely and resulted in product, we can be satisfied with. Some requirements were dropped, some extra features were added over time. Of course we had our share of problems but we certainly were able to handle them.

Our final product consists of a server and some clients. The server-software runs on a Linux-computer with a radio- and soundcard. The planner module controls the radio card, the audio module controls the soundcard and the buffering, the streamer module sends audio data over a network to the clients and the interface module handles client commands.

Contents

Chapter 2

Introduction

This year we were able to participate in the Design project (’Ontwerpproject’ in Dutch). The goal of this project is to bring theory in to practice and to get familiar with the entire process of creating a software application. A list of possible assignments was available and we picked a few nice ones from it. In the end however we took another assignment: ’rivo’. This was a proposal of our accompanists. They gave us the choice between our primary choice and rivo. Although rivo appeared to be harder and to require more Linux knowledge we saw more challenge in it.

The name rivo comes from Tivo. Tivo is a commercial digital video recorder, supporting time-shifting, user-profiles and semi-intelligent automatic recording. Our assignment was to build comparable system for radio instead of tv. In this report we will describe how we tackled ’the rivo problem’.

Many thank go to the following people:

- Our accompanists: Pierre Jansen and Hans Scholten
- The people behind Icecast, Lame and all the open source tools we used
- Barry’s girlfriend Martine, for checking parts of the report

Chapter 3

Goal

To digitally record radio programs and the direct or delayed playing of recordings. Using rivo over a network is desirable.

Chapter 4

Requirements

Derived from the goal we came up with the following requirements:

Recording: The system must be able to record from a desired channel at a desired moment.

Playing: The system must be able to offer recordings and the user must be able to play the recordings at different locations and different computers. If possible the playing must be able to be paused, rewind and forwarded.

Listening to the radio: It must be able to listen live to radio broadcasts. It must be possible to temporary interrupt the broadcast. Then the audio is stored, so that there can be listened later to that audio. Because of this the broadcast isn't missed. The playing of the missed part of the broadcast can be done while the broadcast is still broadcasted. The broadcast is now played with a delay.

User interface: The system shall have a web interface. Because then the user can control the radio with their favourite browser. On the internet there are online radio guides. The system can parse these guides and use them, so that the user can easily select his favourite programs.

User profiles: It must be able to make a profile, so that the system can decide for itself which programs it has to record. This procedure can be complex, but we keep it simple.

Chapter 5

Design

After we gathered all the requirements, we started with the top-level design. We did this by specifying clearly the separate tasks of the system.

In the first design we separated the system into 4 major parts, namely:

1. The streamer, this part would be responsible for the transport of the audio stream over the internet.
2. The audio module, this part would be responsible for the time shifting features.
3. The hardware control, this module would provide a set of functions to control the hardware devices.
4. The user interface, this module would take user commands and control the system according to them.

After we made this design we divided the responsibility for the modules among the group members and started to do some research on the related topics. Soon we found out that this design had some flaws. Hardware control for instance was a very small module with very few tasks and another even bigger flaw was the fact that in this design there was no module responsible for the scheduling of the programs. Therefore we continued working on the design and came up with something better. The hardware control modules was removed and its responsibilities were divided among two other modules. The control of the soundcard was moved to the audio module, and the control of the radio card became a part of a newly added module. This new module was the planner module. The planner would be responsible for the scheduling of programs. This new design proved to be a lot better than the first and it also resulted in a equal division of the work. The final design is shown in figure 7.2.

5.1 Interfaces

After we made this division of work we could finish a big part of the implementation. But when the modules were nearing completion we had to think about how we were going the join the modules together; we needed to design interfaces between the modules. For the design of the modules we used the following approach: At first we made a list of requirements each module had for the other modules, when we had this list we translated the requirements into function headers which should be used in the code. This approach proved to work quite well. Most of the interfaces were rather small since the modules had clearly separated tasks, the interface between the planner and the user interface however proved to be quite extensive, this was not the result of a flaw in the design but of the fact that a user command almost always has an effect on the planner.

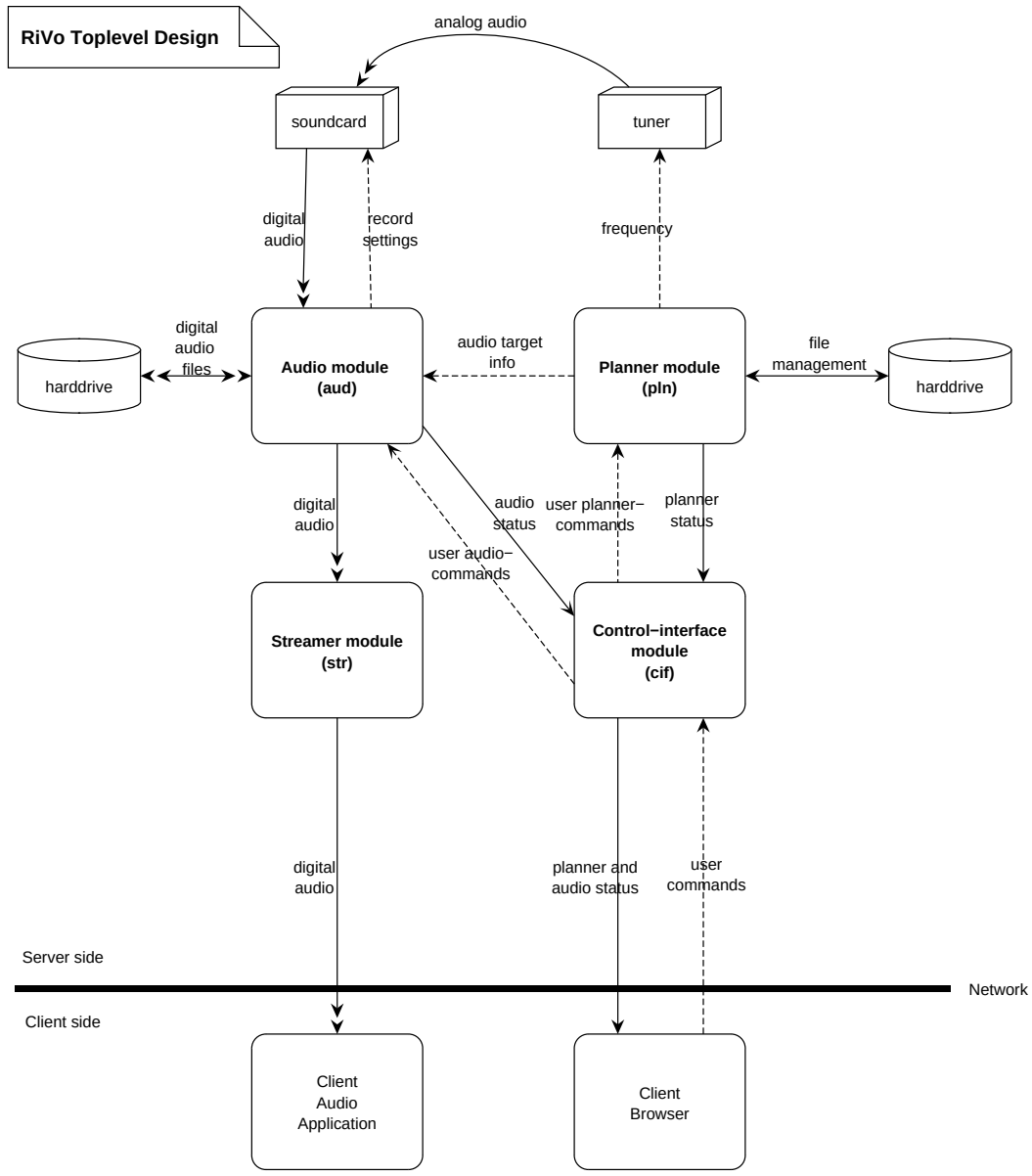

Figure 5.1: Toplevel design

Chapter 6

Implementation

After setting up the initial design of rivo, we started looking at the implementation options. This chapter only describes the top-level implementation issues. Details about modules will be discussed in the following chapters about the individual modules.

6.1 Operating system

First we decided that the server application should run on a Linux system, because Linux easiest, cheapest and most adaptable operating system to program in.

6.2 Programming language

We had to decide in which programming language we were going to build the system. There were 3 serious options: c, c++ and java. Java is an interpreted language and therefore it is too slow to build a "real-time" streaming audio application. Furthermore, Java is not meant to interact directly with hardware. So there were 2 options left: c and c++. Since we were familiar with c but not with c++, we chose c as our development language. Another advantage of c (and c++) is that many open-source libraries are available on the internet. This has saved us lots of time.

6.3 Multiprogramming

From the beginning it was clear that the application needed to be multi-programmed. For multiprogramming in Linux there are 2 main options: threads and processes. We chose to use threads, because communication between threads is a lot easier than between processes. The modules in our program need to communicate a lot with each other, so using processes imposes unnecessary overhead. There are several thread-libraries available for Linux, but we took pthreads because this is the most common and most portable standard. We already had some experience with pthreads from the course "Operating systems".

6.4 Handling debug output

One of the first modules we implemented was a debug module. This module handles all debug messages and logs them to a file. Functions from this module can be called from the other rivo modules. It has a priority system and timestamps are added to the output. In the first place, we implemented this module for consistency in the debug output. Otherwise, each module would probably print all its debug output in its own way to the standard output. This would become a real mess. The debug module solves these problems.

6.5 Main module

The last module we built was the main module. This module has the following functions:

1. It reads the main configuration file.
2. It handles signals properly.
3. It starts all the other modules appropriately.
4. It waits until it receives a stop signal.
5. It closes all modules.

The data in the configuration file is passed on to the corresponding module as it starts.

6.6 Rivo states

Once rivo is running, its functionality follows this state-machine:

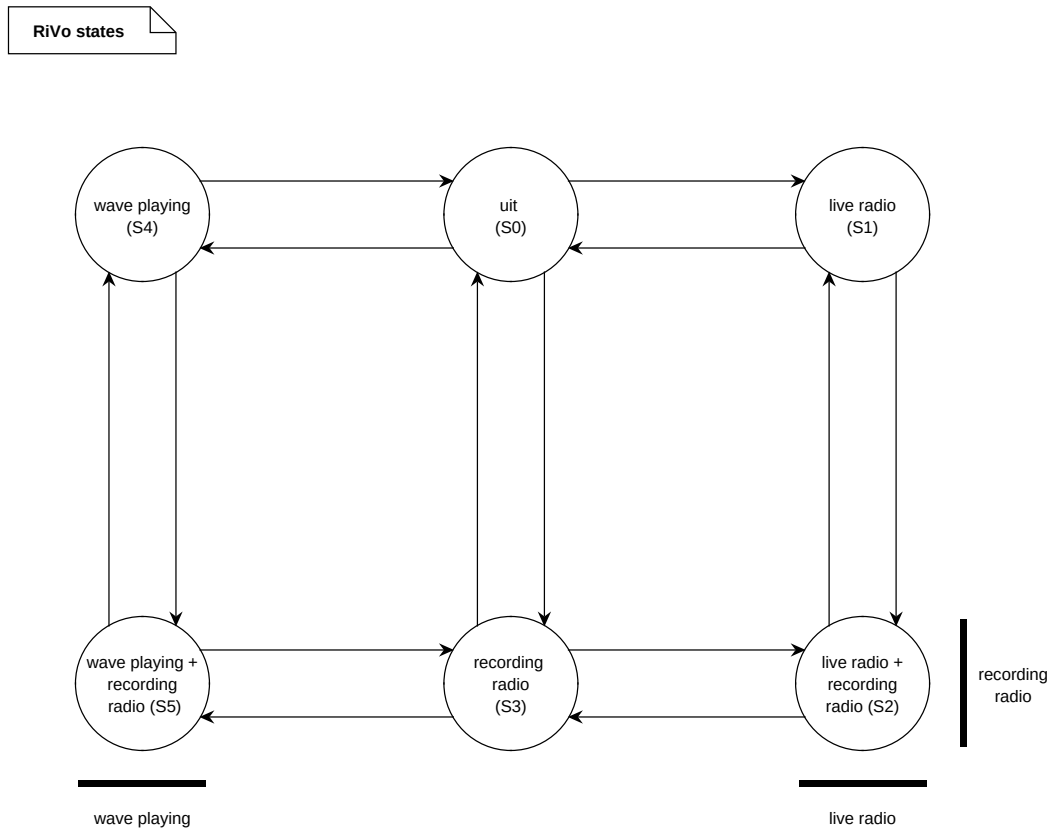

Figure 6.1: Rivo states

In principle this state machine is the product of 3 parallel state machines:

1. Live radio on/off.
2. wave playing on/off.
3. recording radio on/off.

This would result in a 3 dimensional state space, consisting of 8 states (a cube). However, there is 1 restriction: live radio and wave playing can't be on at the same time. This eliminates 2 states, resulting in the state-machine above.

Chapter 7

The audio module

7.1 Goal

The audio module takes care of the audio streams within the application. It reads input from the radio card, handles time shifting, writes audio to disk, reads audio from disk and finally sends it to the user.

7.2 Requirements

The requirements on for the audio module can be summarized in these points:

1. The audio module should read audio from the radio card
2. It should handle time shifting
3. It has to write the audio that comes from the radio card to disk in wave-format when the user wants to
4. It has to play the audio on disk when the user wants to
5. The output has to be sent to the user in 2 ways: through the local speakers or through the network (mp3-stream)

7.3 Design

7.3.1 Design of audio module

The audio module consists of 8 components:

Master input plugin: This input plugin is the main audio source of the application. The audio stream that comes from the radio card is handled by this plugin.

Input transporter: This component is an instance of an audio transporter. It takes care of the input generated by the master input plugin. When necessary, it sends the input to the "disk output plugin" and/or to the "live buffer".

Disk output plugin: This component writes its input to a specific file on disk.

Disk input plugin: This component reads its output from a specific file on disk.

Live buffer: The live-buffer is the component that takes care of the time shifting of the audio stream. Time shifting means that there is a certain delay between the input and the output. The delay may be a few seconds or some hours (it depend on the size of the live-buffer).

A time shift arises when no output is read from the buffer (for example when the output of the main application is paused), while input continues to be written into the buffer. The input data is then stored in the buffer, until reading starts again.

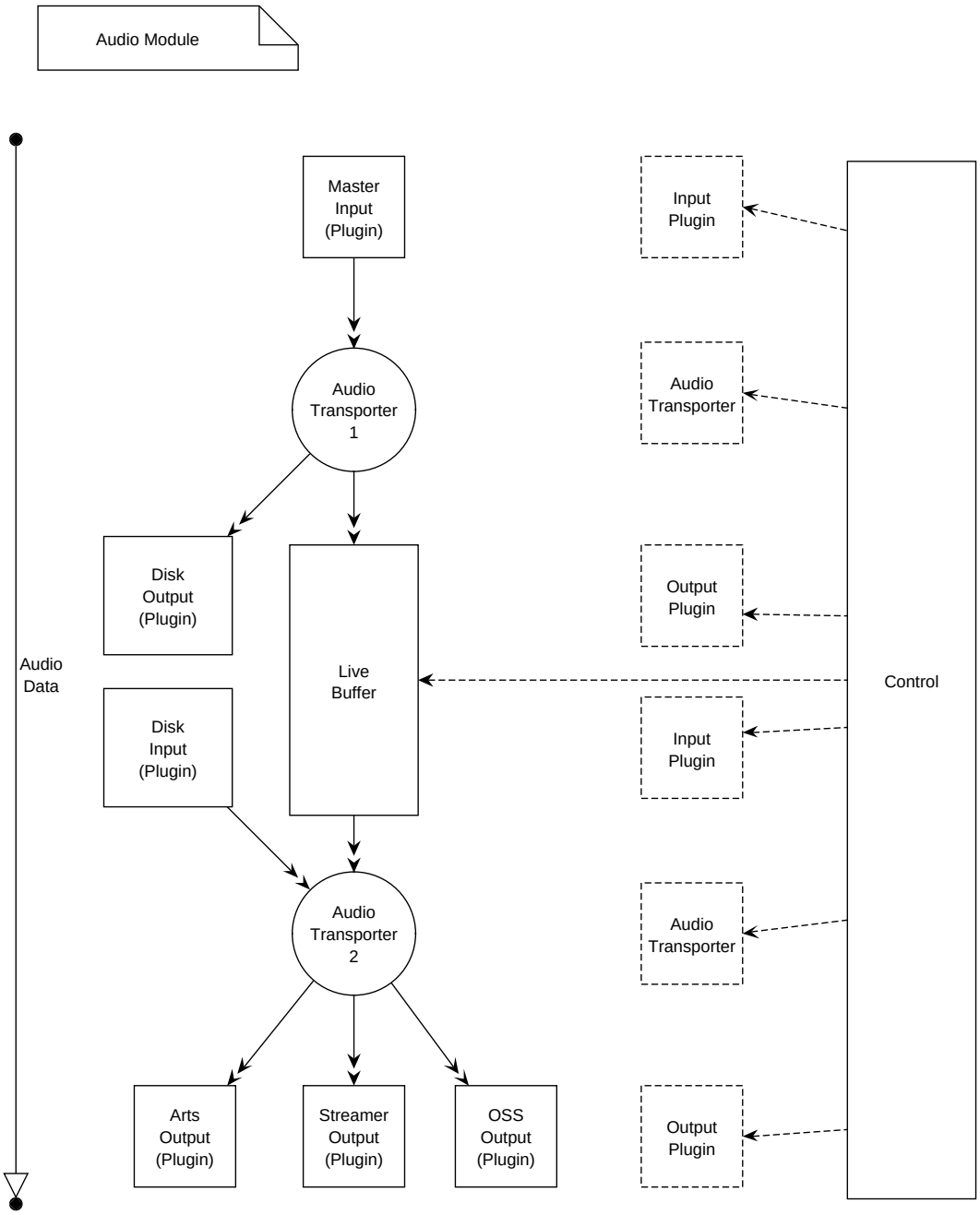

Figure 7.1: Toplevel design

The live buffer automatically stores all recent data that flows through it. The data is kept until the space is needed for other things, like newer data or a time shift that has to be stored. This feature enabled the user to rewind in the recent history. This is another situation when a time shift can arise.

Output transporter: This component is an instance of an audio transporter. It reads its input from the live-buffer or form the disk input plugin. The output transporter sends the audio data to one or more output plugins.

Output plugins: The output plugins are the plugins that send the audio data to the user.

Control: This component forms the interface to the other modules. It also controls all the other components in the audio module.

7.3.2 Design of buffered audio transporters

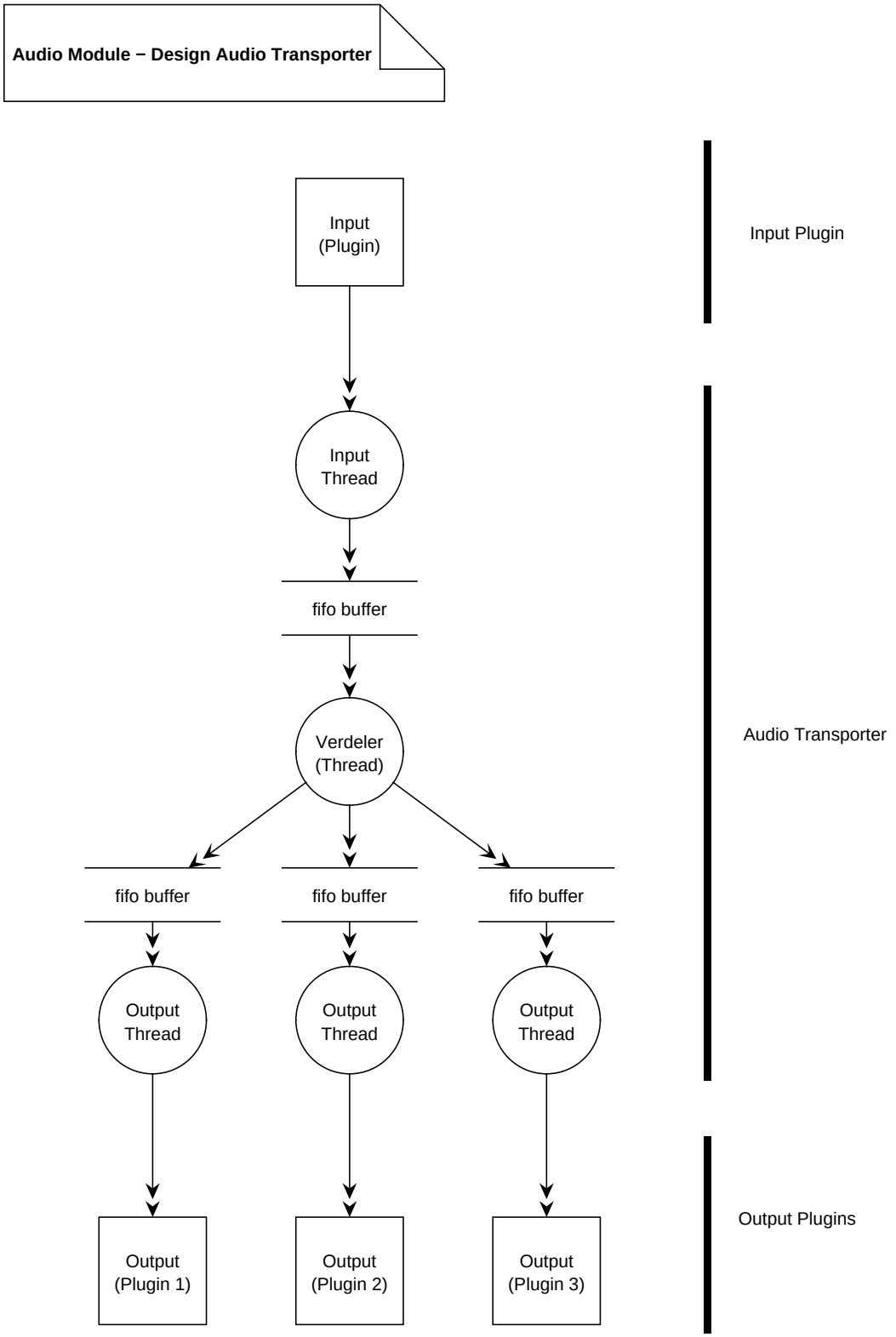

Figure 7.2: Toplevel design

Audio transporters take care of the transportation of audio between input plugins and output plugins. Audio transporters have 0 or 1 inputs and 0 or more outputs. When there are not inputs or outputs connected,

the audio transporter actually does nothing and waits until an input or output plugin is connected.

When there is an input plugin connected and there are 1 or more output plugins connected, the audio transporter reads data from the input plugin and sends the data to all output plugins. The block size of the audio transporter is run-time adaptable.

If an output plugin requires extra buffering or a special block size (for example the stream output plugin), the audio transporter can be configured to meet these requirements: it will create a buffer (the size can be configured dynamically) and it will start an extra thread that reads from the buffer and writes to the output plugin. The size of the blocks that will be read from the buffer and written to the output plugin can also be configured dynamically. The audio plugin will then write the audio data into the buffer instead of writing it to the output plugin directly.

This mechanism of buffering is also available for input plugins. For example: when input is read from the live-buffer, buffering is needed, because a hard disk can't always be expected to deliver the audio data real-time.

7.4 Implementation

7.4.1 Plugins

Plugins are shared objects (.so files) that are compiled separately form the main application. In run-time, they are dynamically loaded into the application.

The advantage of a plugin system like this is that plugins can be added to the system while no recompilation of the application is required, so it makes it more modular and more adaptable.

There are 2 kinds of plugins:

1. Input Plugins
2. Output Plugins

7.4.2 Input Plugins

The task of an input plugin is to provide audio data. It has 3 functions: init, read, close. The meaning of init and close are trivial. The function read reads blocks of data and passes them to the entity that called the read function. An input plugin may have timing, but this is not always the case.

In the current system, the following input plugins are used:

OSS input plugin: This plugin reads input from OSS. OSS is the standard for the audio-drivers in Linux. This plugin sets up the Linux audio device by setting up the sampling rate (22050Hz/44100Hz), the number of channels (mono/stereo) and the number of bits the audio is sampled on (8/16). Then it opens the Linux audio device (/dev/dsp) for reading only. If the mixer is set up properly, the audio that comes in on the line-in of the soundcard is read. The data that is read from this plugin comes directly from the device. This plugin has timing, because the oss kernel drivers are timed.

In the current configuration, this plugin is the master input plugin.

Wave input plugin: This plugin opens a specified wave-file and reads the audio-data from this file. This plugin does not have timing. In this version of rivo, this is the disk input plugin.

Both the wave input and the wave output plugin internally use libaudiofile. This library takes care of reading/writing form/to wave files. There were some alternatives: libsndfile and sndlib. We chose libaudiofile after all because this lib is ported from Irix by Sgi itself and because this lib is used in some large open source projects: EsounD,Alsaplayer and Arts.

Null input plugin: This plugin actually does nothing: when read it called, it immediately return an array with zeros. An array of zeros is silence. Of course this plugin has no timing.

7.4.3 Output Plugins

The task of an output plugin is to accept audio data (and do something meaningful with it). An output plugin has 3 functions: init, write, close. The write function takes a block of audio data and writes it to the target. The meaning of the functions init and close hopefully will be clear. Some output plugins have timing, some don't.

In the current version of rivo, these output plugins are present:

OSS output plugin: This plugin writes output to OSS. It sets up the audio device (/dev/dsp) and opens it for writing only. All data that is written to this plugin is directly written to the device. This plugin has timing.

Stream output plugin: This plugin is the streamer module that is covered by another chapter. This output plugin has timing, too.

Wave output plugin: This plugin opens a specified file for writing (or creates a new one if necessary). All audio that is written to this plugin is directly written to the wave file. In the current configuration, this plugin is the disk output plugin. The wave output plugin doesn't have timing.

Arts output plugin: This plugin writes all data it receives to the arts deamon. Arts is the audio-server of kde2. This plugin has timing.

Null output plugin: This plugin behaves live a black hole: anything that comes near disappears into it and is never seen again. Black holes don't have timing, so this plugin doesn't either.

7.4.4 Audio Encoding

Internally, the audio module always works with the same audio format: pcm (raw). It can switch between 22050Hz/44100Hz, mono/stereo, 8bits/16bits. So virtually all pcm formats are supported. Only 48kHz pcm is not supported. All input plugins generate audio in the pcm format and all output plugins accept audio in the pcm format.

7.4.5 Live Buffer

The live buffer is in fact a synchronized buffer that stores its data on the hard disk. When the live-buffer is initialised, it receives the name of a file that it can use to store the data. The size of the file is automatically detected. It is the responsibility of the user to create a file that is large enough.

There are 2 threads involved in the live-buffer. One thread writes data into the live-buffer, the other thread reads data from the live-buffer.

If the buffer is completely filled (so the time shift is at its maximum size) and the writing thread wants to write audio into the buffer, the data is thrown away and the write function immediately returns. So the write call for the live-buffer is not blocking.

There were 3 alternatives for handling the audio that is written when the
buffer is completely filled:

1. make the write call blocking

   This is the common way to implement a bounded buffer, but for this application it is not useful, because when the writing thread is blocked, it can't read from the audio device, so the buffer of the audio device will be full in some time. When the buffer of the audio device is full, the Linux kernel will throw away any new audio data. So after all new audio will be lost anyway.

   Another disadvantage of the blocking write call is that the writing thread gets blocked, even when it doing something useful besides writing in the live-buffer. For example: there are situations when the input audio transporter is writing in both the live-buffer and the disk-out plugin. If the threads gets blocked by a write call to the live-buffer, the writing to the disk-out plugin also stops. Of course, this is not desirable

2. don't make write call blocking but overwrite the oldest data in buffer

   With this alternative the writing thread doesn't get blocked so it will never stop writing to disk. But when the buffer is completely filled, the time shift takes up all the buffer, because the user pressed the

pause button in the past. When you overwrite the oldest data in the buffer, you overwrite the audio that was recorded just after the user pressed the pause button, but probably that was important to the user (why else would he press the pause button?). Probably the audio that is recorded at this time is less important to the user than the audio that was recorded just after he pressed the pause button. So overwriting is not an option.

3. throw away any audio that doesn't fit in the buffer

   This alternative has the same disadvantage as the first alternative (audio gets lost), but it solves the problems of the first 2 alternatives. We chose this alternative for the implementation of the live-buffer.

If the buffer is empty, the reading thread is blocked until enough data is available. There were 2 alternatives for handling read calls when the buffer is empty:

1. return a block of silence (=zeros)

   Returning a block of silence might be an option when the it is possible that the reading thread calls the read function when the buffer is empty and when it is not guaranteed that new data will be written in the buffer in a short time. However, the control component guarantees that when a thread is reading from the buffer, another thread is writing in it.

2. block the reading thread until enough data is available

   In theory it shouldn't make a difference if the read function blocks the reading thread or not when the buffer is not entirely empty. In practice it sometimes does.

   For example: when the buffer is almost empty and a output plugin gets connected to the output transporter that has a large internal buffer (for example the stream output), the buffer will be filled at a very high speed and if the buffer of that output plugin is bigger than the amount of data in the live-buffer, then the rest of the buffer will be filled with silence. This causes hicks in the audio stream.

   Because the control component guarantees that when a thread is reading from the live-buffer, another thread is writing into it, it is safe the make the read function call blocking. There will always be audio available in a relatively short time. In practice we experienced that this solution doesn't cause any hicks at all.

7.4.6 Audio transporters

In the design section the design of the audio transporters is explained, so in this section some interesting implementation issues of the audio transporters will be covered.

The audio transporters treats every input source or output target as a plugin. This is done by calling function pointers to the read and write function of the inputs and outputs. When for example the live-buffer is added to the input transporter as an output, the pointer to the write function is passed to the audio transporter. This way the audio transporter can't see the difference between a plugin and the live buffer and this makes the audio-transporters more flexible.

If for some reason an input or an output needs extra buffering (live-buffer), or a special block size (stream-out), then the only thing you have to do is to change the configuration file an then the audio transporter puts an extra buffer in between. We implemented this buffer ourselves and named it "universal buffer". This universal buffer is just a bounded buffer (like a pipe) but it can be initialised at any size you like. Its implementation is very efficient because it uses memcpy.

7.4.7 Audio Module Control

The control component forms the interface of the audio module to the other modules. It is completely synchronized. It receives commands from the cif module and from the pln module.

init(): This function initialises the audio module. It reads the configuration file, it loads and initialises the master input plugin, it loads the disk in and disk out plugins, it initialises the audio transporters, it connects the null output plugin to the input transporter and it connects the null input plugin to the output transporter.

close(): This function stops all threads, it closes and unloads all plugins.

start_recording(char *filename): This function starts writing the input to a file. It initialised the disk out plugin and connects it to the input transporter.

stop_recording(): This function stops recording. It disconnects the disk out plugin from the input transporter and closes it.

start_playing(char *filename): This function starts playing from a file. If the system is live, live-listening is stopped first. Then the disk in plugin is initialised and connected to the output transporter, but first it disconnects the null input plugin from the output transporter.

stop_playing(): This function stops playing from a file. First, it disconnects the disk in plugin from the output transporter. Then it closes the disk in plugin and it reconnects the null input plugin to the output transporter.

start_live(): This function starts live-listening. First it disconnects the null input plugin from the output transporter. Then is connects the live-buffer to the input transporter and the output transporter.

stop_live(): This function stops live-listening. It disconnects the live-buffer from the input transporter and from the output transporter. It also empties the live-buffer. It also reconnects the null input plugin to the output transporter.

pause(): This function removes the current input of the output transporter and connects the null input plugin to the output transporter.

resume(): This function removes the null input plugin from the output transporter and reconnects the original input.

7.4.8 Configuration File

Many properties of the audio module can be adapted by changing the configuration file. After the changes are made a restart of the program is needed.

For each inputplugin, there is a section in the configuration file. The buffer size and block size can be changed here. When the buffer size is 0, there is no extra buffering and the value for block size is ignored.

For each output plugin there is a section in the configuration file. The fields have the same meaning as for the input plugins. If the field selectable=1, the output plugin can be used as main output plugin (in the current version, there are 3 man output plugins: oss_out, stream_out and arts_out). If the field selectable=0, then the output plugin can't be used as main output plugin.

7.5 Problems/Alternatives

7.5.1 Clients are kicked when there is no audio stream

When clients are connected to the Icecast server and the encoder stops sending data, the clients are kicked. Originally, the audio module stopped sending data to the output plugin when it was paused or when the system was idle. We solved this problem with the "null input plugin". This plugin is always connected, unless there is actually read audio to send (when it is playing from a file or when live-listening is enabled).

7.5.2 Bugs/Features of OSS

Full duplex soundcard drivers

When we tried to use both the oss input and the oss output plugins on an computer with an SoundBlaster AWE64. This didn't work, because for this soundcard the old SoundBlaster 16 drivers were used, and these drivers only support full-duplex sound processing when the device-file is opened as read/write.

In our modular system, oss input and oss output are handled by different plugins, so we have to open the device 2 times: one time as read-only and one time as write-only. This was not possible with the old SoundBlaster 16 drivers, so a new SoundBlaster 128 PCI (chipset ES1371) was installed. The drivers for this card allow a device to be opened twice, so our problem was solved.

With the current plugin system that strictly separates the input and output plugins, we couldn't have solved this problem in software. A more sophisticated plugin system is needed for this.

Strange buffering

Another problem we experienced with the oss kernel drivers is that the buffer in the audio device sometimes contains old data. For example: when you start listening to the radio, then you stop listening, then you wait a while and then you start listening to the radio again. Then you first hear about half a second of the music that was on the radio when you stopped listening and then the music that currently is on the radio.

This is caused by the buffer of the device file that is not updated when the data is not read. When we stop listening to the radio, the oss input plugin is not closed, this means that all the time the file is opened and apparently the audio drivers of the Linux kernel don't overwrite or flush the buffer.

We fixed/worked around this problem by adding a null output plugin that is always connected to the input transporter. What happens then is that even when you are not listening to the radio, the radio input is continuously read (and thrown away in the null output plugin). Due to this the buffer of the audio device is always up to date. This workaround takes approximately 0.1

7.5.3 timed plugins versus non-timed plugins

Some plugins have timing. This means that they won't accept/produce data faster than it can be put through your speakers. Some plugins don't have timing, so they accept or produce data as fast as they can.

In the audio module, both timed and non-timed plugins are treated in the same way. Normally this causes no problems, but when new plugins are added to the system you have to watch out a little, because when the input and all the outputs of an audio transporter are non-timed, things go wrong.

For example: when an audio transporter has an input that is timed and it only had non-timed outputs, then the outputs are indirectly timed by the timed input. The audio data just won't flow faster, because the input won't produce the audio faster.

Another example: when an audio transporter has an input that is not timed and it has some outputs that are not timed and one output that actually is timed. Now the data flow in this transporter is indirectly timed by the one output that has timing. The input and all the outputs can produce/accept the data faster, but that one output plugin with timing slows up the audio stream, with the result that the whole audio stream is timed.

Things go wrong when an audio transporter has no timed inputs or outputs at all. Then the input produces audio as fast as it can, the outputs accept data as fast as they can, the processor use will go to 100

A solution for this problem is to give audio transporters a timing mechanism for themselves. Usually this is overkill, because most plugins have timing for themselves. It also makes the audio module less adaptable, because when the audio transporters have timing, they have to be aware of what type of data they are transporting. Another solution is to add timing to every plugin that doesn't have timing, or at least to the plugins where timing is needed.

In the current implementation of rivo, we have not adopted any of these solutions. We rely on the timing of the input and output plugins for the timing of the entire audio module. This works, because the master input plugin (oss input) is timed and all the main output plugins (oss-out, arts-out, stream-out) are timed. Because of this the input audio transporter and the output audio transporter always are timed.

7.5.4 Limitations of plugin system

The current plugin system has some limitations:

fixed number of arguments for initialisation of plugins

When a plugin is initialised, it takes a fixed number of arguments, even when the plugin doesn't need all arguments. For example: the stream output plugin ignores the target initialisation argument, but it gets it anyway, because there is an uniform interface for all output plugins. Actually there are some plugins that need the target argument, so all plugins get this argument.

On the other side, some plugins have parameters that are not passed at initialisation time, so these parameters are compiled into the code. For example: the bit rate of the mp3-stream could be run-time adaptable, but this parameter is not part of the uniform interface for the output plugins, so this option is not passed when the stream output plugin is initialised.

To solve this problem, either every output plugin would need to read its own configuration file or the number of arguments would have to be variable. The first solution is not a very nice one, because every

plugin would need its own configuration file parser and all those small configuration files would turn your hard disk into a mess.

The second solution would make the plugin loading and initialisation routines complicated, but the plugins remain very simple. Other projects handle plugin/driver arguments in a similar way, for example the XFree86 project.

separation between input and output plugins

In the current plugin handling there is a strict separation between input plugins and output plugins. This has one very important advantage: it keeps the plugin handling and the plugins themselves very simple. It also has a great disadvantage: plugins can't be input-plugin and output-plugin at the same time. This implies that for example the live-buffer couldn't be implemented as a plugin. Of course, this limits the adaptability of the audio module.

The obvious solution for this problem is to build a plugins system that also supports input/output plugins. Or maybe even plugins with multiple inputs or outputs might be useful, for example if you want to build mixers or faders. But before we start with a project like that, we might want to take a look at projects like Artsbuilder or Glame, just to be sure that we aren't building something that has been made before.

With a plugins system that supports input/output plugins, it would have been possible to make a combined input/output plugin for the old SoundBlaster 16 kernel drivers, so that we could work around the limited full-duplex support of these drivers.

no status information can be transferred

In the current plugin system, no status information can be transferred from the plugins to the system. For example: when the system calls read on a plugin, the plugin returns the data or - when no data is available - it blocks the thread until enough data is available. Output plugins behave in a similar way: when a write is called and there is no space available to store the data, the thread is blocked until the space is available.

As long as input plugins always have data (or when they will have data in a relatively short period) and output plugins always have space (or when they will have space in a relatively short period), this is not a problem. As long as plugins are guaranteed to return form read or write calls (how long it takes actually doesn't matter), there are no problems. But when plugins cannot guarantee that they will return (how long it takes doesn't matter) from a read or write call, problems arise.

For example: if the live-buffer were just a synchronized bounded disk buffer (actually it is not, but let's say it is) and there were no data in the buffer and a read was called, this function call would block until there is enough data to return. If no other thread is writing into the live-buffer, the reading thread would be blocked forever. To itself, this is not that bad (there is no audio data anyway so why should the not be blocked..), but when the thread has to be destroyed real problems (deadlocks) arise.

There is an obvious solution to this problem: use asynchronous thread cancellation. But using asynchronous thread cancellation would require that every function that could be called by a thread has a cleanup handler. But cleanup handling in combination with synchronization is really very hard to implement (if possible at all). This would result in more complicated plugins and that is not desirable.

Another solution is to block threads in the audio-transporters instead of in the plugins. This would make life really easy: no cleanup handlers would be needed and threads can be unblocked whenever we like to (for example when they need to be destroyed). However, this solution requires that the audio-transporters can ask the plugins for their status (for example they need to know how much data is available in input plugins). A mechanism for notifying the audio-transporters (for example when new data is available) would be needed too.

7.6 Testing

In order to test the audio module, we wrote a special program called aud_test. This program used a primitive keyboard-interface, but it was functional enough to discover problems/bugs/deadlocks etc. Because the audio-module depends on no other modules, testing was pretty straightforward and bugs were usually easy located.

7.7 Hooks/Future

During design and implementation, we tried to keep the design and the implementation as flexible and adaptable as possible. The result is that in the future, without too much effort, some features can be added:

7.7.1 Extra formats for audio-files on disk

Currently all files are saved in the pcm-wave format. This format has no compression. If you would like to save the files in a compressed format, this is possible. Just write 2 plugins (one for input and one for output), change the configuration file and it works.

7.7.2 Extra audio sources for the application

At this time, the program takes its input from the soundcard. However, if in the future a different audio-source is needed, this is possible, because the input from the soundcard is read by a plugin.

7.7.3 Extra output plugins

In this version, there are 3 main output plugins: oss-out, arts-out and stream-out. If in the future another output plugin is needed (for example an ogg-vorbis streamer), this is easy: just write a plugin, add a section to the configuration file and rivo can send ogg-vorbis streams.

7.7.4 Entirely different data formats

Rivo was originally designed and implemented for radio. However, people seem to be interested in a version that supports television or even all types of streaming media. This feature is not easy to implement, but we think it can be done.

The live-buffer and audio transporter components don't know what type of data they are buffering/transporting. They don't see the difference between a pcm-audio stream or a mpg2-video stream. The only components that are aware of the type of data are the plugins and the control component (that controls the audio module and that forms the interface to the other modules). These components would need to be re-written. This is a large job, which certainly is not easy, but, as said before, it can be done.

Chapter 8

The streamer module

8.1 Goal

Multiple clients must be able to listen to rivo at the same time over a network.

8.2 Requirements

1. Multiple clients must be able to listen to the radio output at the same time.
2. The audio must be presented in mp3 data to the clients.

8.3 Design

The streamer gets wave audio from the audio module and gives it in the right format to a mp3 encoder. The encoder returns mp3 data to the module. After the encoding, the streamer module sends the data to a program that handles the streaming of the mp3 audio. Multiple clients can connect to that service.

8.3.1 Design decisions

There were two major design decisions made for the streamer module: The decision of an own streamer vs. icecast and the decision about the usage of 8 bits audio data.

Own streamer vs. icecast: At the beginning of the project we thought of making an own server instead of using icecast. This idea came partially because we thought that icecast decodes and re-encodes mp3, so that icecast was very inefficient. But after some research we concluded that icecast didn't decode and re-encode the mp3 data. An own streamer also had some problems/difficulties. The greatest difficulties were that the protocol icecast uses (the icy protocol) was nowhere to find and that it was more work than we thought (handling things like sending and timing). With the knowledge that icecast and libshout could do the job well and in a shorter period, we decided to make an icecast version first (with icecast and libshout) and after that we could consider making our own streamer. When the icecast version was almost ready and time was short, we decided to use the icecast version, because we had not much time left and icecast handles the communication well.

8 Bits audio data: We began with the idea of making a streamer module that could handle 8 bits and 16 bits audio. The developing and testing of the streamer module and audio module made clear that 8 bit sound was really bad. With that conclusion we decided on February the 25th 2002 that we didn't pay attention to 8 bits audio anymore. The option remained in the program, but not optimised.

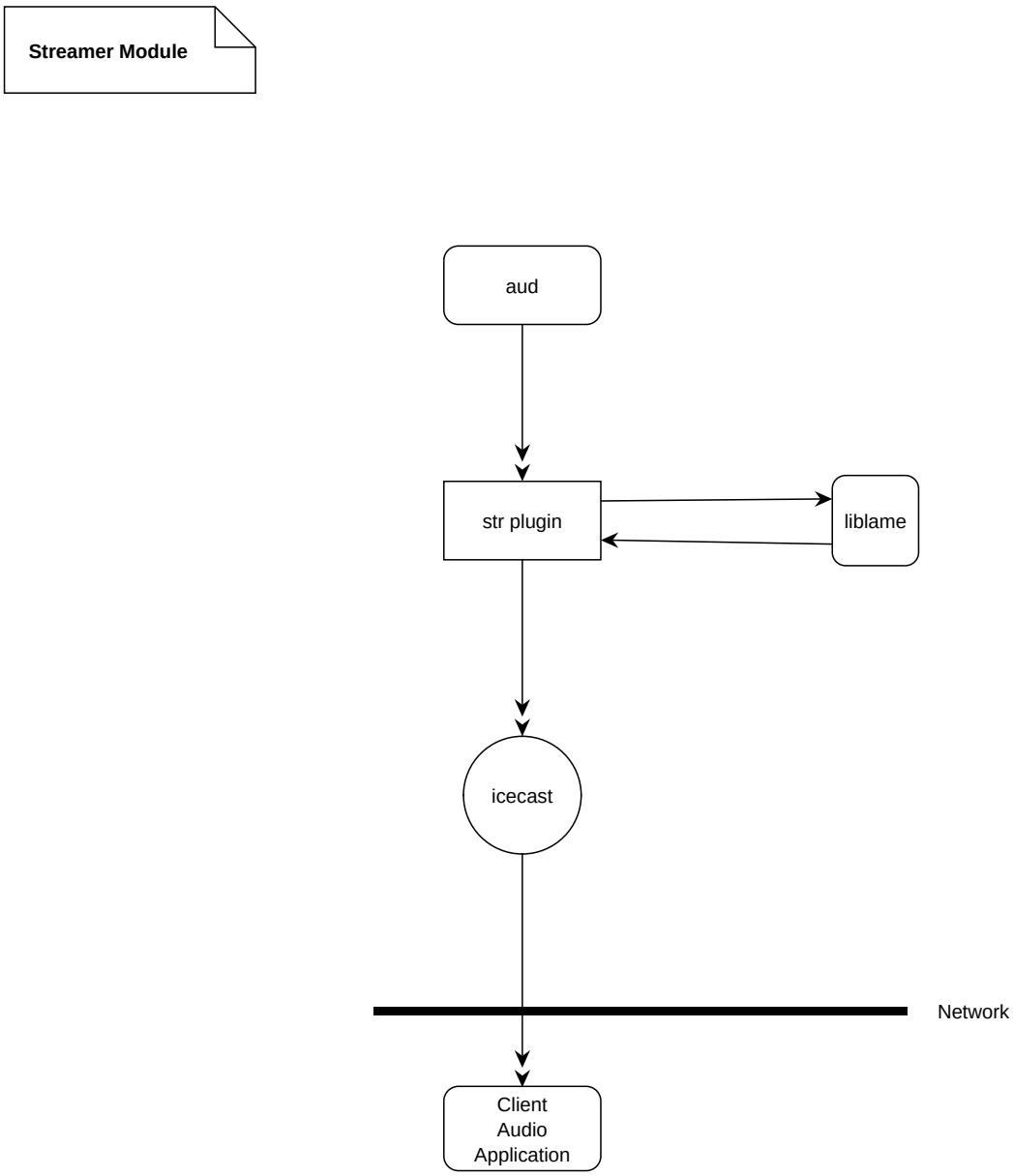

Figure 8.1: Streamer design

8.4 Implementation

8.4.1 Programs used

The streamer module uses three external (developed by other people) applications. These are: icecast, libshout and LAME. Icecast is an application that handles the multicasting over a network (i.e. internet). Because of icecast the module can send data to multiple clients. Libshout is a library that handles the communication between the module and icecast. This library is needed to get real-time streaming working. Icecast itself can only handle files, but libshout has the ability to send buffers of mp3 data to icecast. LAME is an application that encodes pcm (raw) or wave audio data into mp3 data. The streamer module uses libLAME, that is the library function of LAME without the front-end application. This library gets one or two buffers of pcm data and converts it into mp3 data and puts it into a buffer.

8.4.2 Implementation

The streamer module is divided into 3 subsections (like all plugins): initialisation, write, close. The implementations will be explained in that order.

initialisation: In this stage of the plugin the connection to icecast is initialised using libshout. The module also initialises LAME at this section. It sets the quality, mode, compression and sample rate of LAME. These settings are set according to the given (by the audio module) sample rate, number of channels and number of bits. Other settings of LAME are set to a default value chosen by LAME. After this procedure the module is ready to write to icecast.

Write: When the audio module calls the write function of the streamer module, the given buffer is copied to a temporary buffer. After that the temporary buffer is split into pieces of short integer buffers. Short int's are used, because LAME wants short ints and the audio is 16 bits at most (length of a short int). The procedure of splitting goes like this: The temporary buffer is a buffer of bytes. This buffer is chopped in pieces according to the number of channels and the number of bits used for the data. There are four options.

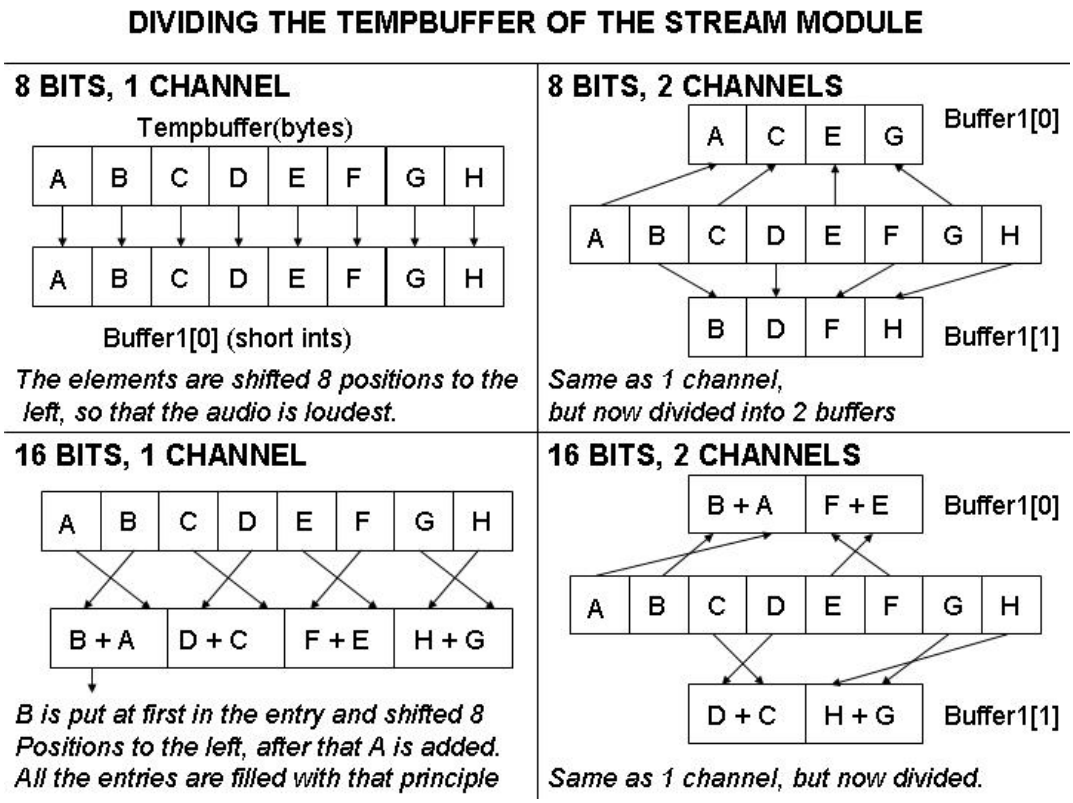

Figure 8.2: Dividing the buffer

1. 8 Bits and one channel: one element from the temporary buffer is one element in the buffer to be send to LAME. Because 8 bits are one byte and there is only one channel, therefore one sample of audio is 1 byte long.
2. 8 Bits and two channels: one element of the 2 buffers given to LAME represents two elements in the temporary buffer.
3. 16 Bits and one channel: two elements of the temporary buffer represents one element at the LAME buffer (1 buffer).
4. 16 Bits and two channels: four element of the temporary buffer represents one element at each LAME buffer (2 buffers).

Further explanation about the separation of the data can be found in 8.2. After the dividing into one or two buffers for LAME, the buffer(s) is/are send to LAME, that encodes the buffer(s). Next the internal mp3 buffer is flushed into the mp3 buffer used by the module. If the encoding is done correctly, the data is send to icecast using libshout. Finally the shout connection will wait until the mp3 data is played (a prediction). This function handles the timing for the connection to icecast.

Close: All the buffers of LAME are released and the connection to icecast with libshout is closed.

8.4.3 Problems

One of the problems encountered during the construction was the separation of the temporary buffer. At first it wasn't plain what the size of the parts were, i.e. which channel comes first and which byte of a 16 bits sample comes first. But pretty soon, after a conversation, it was clear how the audio data was send. And when it was clear what was send, it wasn't very difficult. More problems didn't occur while developing the module. Most of the problems occurred during the combining of the modules. This is evident, because the streamer module couldn't be tested thoroughly without the audio module, as the module strongly depends on the audio module.

8.4.4 Difficulties

A difficulty while developing the module was the communication between the module and the used applications. It wasn't very difficult, but there was much to read. The 3 used programs (LAME, icecast and libshout) had to be gone through thoroughly before they could be used. Another difficulty was to separate the buffer into appropriate pieces for LAME. Where the temporary audio buffer has to be cut, depends on the number of bits and the number of channels. A further explanation is available in 8.2.

8.5 Testing

Testing the module isolated from the rest of the program wasn't feasible. The only 2 tests that could be run were compiling the source and testing the libshout connected to icecast. Practically all the testing was done in cooperation with the audio module, because the streamer module can't function without the audio module. Testing with the audio module began with the streaming of a wave file and without a live buffer. These settings were the only settings available at that time and that part worked well. The initialisation phase was tested first. After that part worked ok, the testing of the writing began. The first tests were related to the encoding of the audio into mp3 data. The mp3 data wasn't send to icecast at that point of testing, but was written to a file. At first there came out noise.

After some testing the conclusion was drawn that the 2 bytes of one sample for 16 bits and 2 channels had to be swapped. Instead of putting the first before the last, it had to be done the other way. This is because the computer works little endian. This fact is well known, but not intuitive, therefore the mistake was easily made. When there was good music from the mp3 file, the connection to icecast had to be tested. That test went well. The data was correctly send to icecast. After that, the testing was paused for a short time. Because the audio module wasn't ready to test the streaming of audio data.

When the real time streaming of audio data could be tested a problem occurred. The audio came with bleeps and silences, where it wasn't intended. This problem was solved for a short time: We tuned the block sizes of the output plugin and such. But when we used another configuration (another number of bites, channels etc.) the problem occurred again. After some tuning of the block size again, everything worked. We concluded from this incident that there was something wrong with the block size. And we were right, the block sizes of the buffers weren't guaranteed. When that was fixed, the audio was played correct.

8.6 Future

8.6.1 Future additions

Things that can be added to this module in the future are:

- Storing the mp3 data in files
- Streaming of audio in ogg format (lame supports ogg encoding, still beta)

8.6.2 Improvements

Things that can be improved in the future are that several options are stored in a configfile and/or given to the module by the audio module. If these options are set in a configfile the user can modulate the stream much easier. Options that are convenient to store in a configfile are:

- The compression rate: this option regulates the kilo bits per second (kbps). At this time the kbps is set to 128 in all modes (all varieties of channels and sample rates).
- The quality of the stream: Now the quality can only be changed by editing the code.
- Variable Bit Rate: VBR is an option for LAME to use a variable bit rate. Now VBR is not used, because the computer can't handle that, but the option exists in the header file.

8.6.3 Alternatives

There was only one alternative for the choices made for the module: constructing an own streamer instead of using an external program (icecast). Arguments in favour of an own streamer are:

- It's your own source, so that you know what happens.
- You can optimise it to your own purpose.

Arguments for using external programs are:

- You don't have to invent the wheel again.
- It does what we need and good.
- Time wasn't on our hands.

Time was a great aspect and the external program (icecast) worked good, so we chose for the external program.

8.7 Remaining problems

A remaining problem in the streamer module is that the audio stream malfunctions after a long time. This problem occurs while doing this procedure: First the streamer plugin is enabled, then the clients connect and after that the radio is set to a channel. The other scenarios work well. This problem is not solved, because the problem lies within LAME. We think that when LAME initialises during a silence (self made) the audio properties are set incorrectly. But if you disable the output plugin and enable the output plugin, the stream is correctly again. So when that problem occurs you only have to disable and enable again and your problems are gone

Chapter 9

The planner module

9.1 Goal

The goal of the planner module is to take care of 3 major tasks:

1. To schedule and store user added programs (both one time and repeating programs).
2. To control the radio thread and the audio module according to the user added programs.
3. Store information about the audio recordings resulting from the users programs.

9.2 Requirements

From these 3 main tasks a lot of sub tasks can be derived, these sub tasks are given per major task:

9.2.1 Scheduling and storing programs

1. To be useful there must be a way to add programs.
2. There must be a way to remove programs.
3. To guarantee the feasibility of the program there must be a function to check conflicts between newly added programs and already scheduled programs.
4. To keep the program list clean there must be a way to filter invalid programs (with negative times and stuff like that).
5. To prevent loss of programs in case of a server shutdown the program list must be written to disk.
6. To restore the program list when the server starts there must be a parser to parse a configuration file back to a list.

9.2.2 Controlling of the radio device and the audio module

1. There needs to be a set of functions to control the radio device.
2. There needs to be a well-defined interface with the audio module.
3. There needs to be a function to get the next program.
4. The current program must be cancellable (for the users convenience).
5. Programs need to have a way to indicate on which frequency they are broadcasted.

9.2.3 Storing the information about recordings

1. The list of audio files must be editable so programs can be added and removed.
2. Audio file descriptors must have the path of the corresponding audio file (to link the two together).
3. It must be possible to write the list of audio files to disk.
4. It must be possible to parse the written list back to a normal list.

9.3 Design

The module basically is divided in two main parts: The radio thread and the planner. The radio thread controls the radio device and tells the audio module what to do. The planner handles the program lists and the information about the recorded files. Problems during the implementation of the planner caused a few changes in our original design. At first the idea was to let all the communication between the radio thread and the planner go through a central data store, in this data store would be a combined list of periodic programs and normal programs scheduled a day (or another period of time) ahead. The radio thread would try to read the first program from the store and execute it. The planner would periodically add new programs so the radio thread would never run out of work. That would be about all the communication there was between the radio thread and the planner.

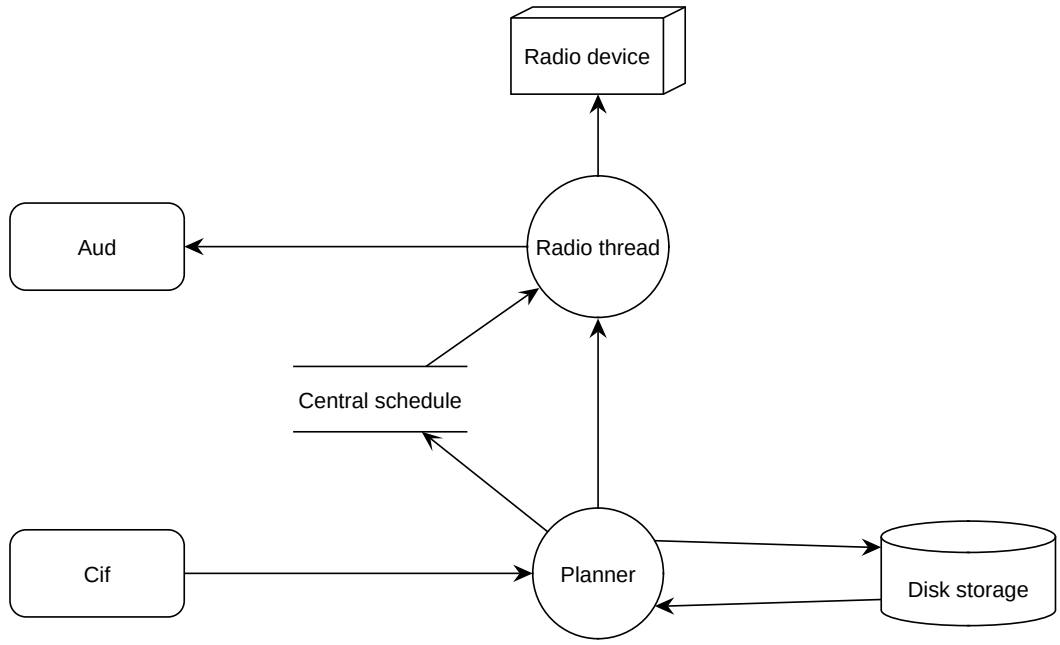

Figure 9.1: Original design of the planner module

At first this design seemed to be nice and modular, yet soon it posed a few big problems:

1. Programs could not be cancelled since there was no way the planner could tell this to the radio thread (except by adding more lines of communication).
2. Storing periodic programs and normal programs in one data store (as normal programs) was very memory inefficient, a periodic program with an indefinite number of periods would also require an indefinite number of normal programs in the memory.

Looking at these problems we came to the conclusion that this strict separation between the radio thread and the planner was not very useful. Therefore we removed the central schedule and replaced it by 2 separate data stores (for programs and periodic programs). In this design the radio thread would just ask the planner for the next program if it needed a new one, in this way the radio would always get up-to-date

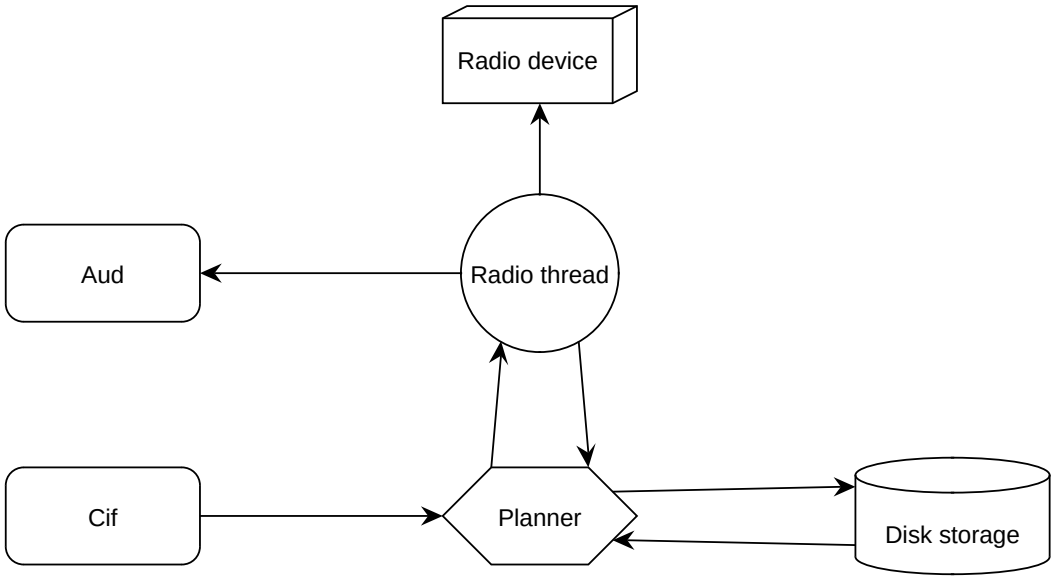

Figure 9.2: Final design of the planner module

information. This would cause far less problems with updating the schedule and also made it a lot easier to make programs cancellable.

9.4 Implementation

9.4.1 General implementation

The radio thread works quite simple, it is one simple loop of control which goes through the following actions: it first asks the planner for the first program. If the planner returns an error (meaning there is no first program) it waits to get signalled (the planner is responsible for this) until there is a first program. If the planner returns a program the radio thread waits for the program to start. When the program starts the thread sets all the devices and modules in the right state and waits for the program to end. When the program ends the thread cleans every thing and goes back to the start. The radio thread also handles the cancelling of programs.

The planner is a collection of functions that operate on a couple of data stores (linked lists). These functions are mainly called from the control interface. The planner has the following data stores: A list of channels, a list of normal programs, a list of periodic programs and a list of audio files. All the data stores are (as told before) linked lists, this is useful because of their dynamic character, with array's there would be a maximum number of items which causes problem if the users for instance wants to add more channels then the maximum number. This problem could be solved with arrays by making the arrays rather large, but this is very memory inefficient. Therefore linked lists are the best option. Every data store has a few basic functions which are not interesting to discuss in detail like the adding and removing of programs. The first data store is used to save all the channels, in this program channels are structs with a couple of important fields:

name to identify the channel in a way which is convenient for the user.

id to give the channel an unique identifier the program can look for.

frequency to tell the program which frequency the radio should be tuned to.

next_item this is the pointer to the next item in the list.

There are a couple of other fields but they are not used at this moment, they can be used in the future to add extra features. None of the functions on this store are very interesting, it's all rather trivial. The functions on the list to store programs are more interesting, next to the trivial adding and removing of programs there

are functions to check conflicts with other programs. This poses some interesting mathematical problems which will be discussed in the section "conflict checking". Much like the list of channels (and the other two lists) the list of data stores consists out of structs. The structs in this list have the following important fields:

name is to identify the program in a user-friendly way.

description is to store additional data about the program.

start_time contains store the time the program starts (0 if it should start immediately).

end_time stores the time the program should end (0 if it continues indefinitely).

channel_id is to tell on which channel the program is.

live says if the program should play live.

record tells if the program should be stored on disk.

status is to check if the program is scheduled or running or other things like that.

next_item the same as in channel.

Storing periodic programs is highly similar to storing normal programs, a lot of functions on this store however are far more complicated. The checking of conflicts between two periodic programs for instance was one of the most challenging tasks in this module. The structs in this data store are almost the same to the ones in the normal program list, except for a few fields to specify the periodic nature of these programs. The fourth and last data store contains the descriptions of previously recorded programs. This is useful for the users to find the files containing their favourite program, the names on the file system are not very user-friendly. The structs in this list contains the following fields:

start_time contains the time the recording started (this can be a different value than the start_time of the related program.

end_time contains the time the recording ended (this can also be a different value than the end_time of the related program.).

channel_frequency tells on which frequency the radio was tuned during the recording of the program.

file_name locates the corresponding file on the file system.

next_item guess what...

The audio file struct also has some fields for future changes in the program.

9.4.2 Conflict checking

To make sure that all the users programs are handled in the right way programs should only be added if they have no conflicts with other programs. Therefore there needs to be a conflict checker of some kind. In our program there are 3 different types of conflicts which will all be discussed here.

1. Conflicts between two normal programs are rather easy to check, you just have to look at two start times and two end times. At first we checked if there was a conflict by looking for overlapping programs. If program A and B overlap one of the following statements is true: a. The start time of A is between the start time and end time of B. b. The end time of A is between the start time and end time of B. c. Program A is completely in program B's time. Or d. program B is completely in program A's time. This check was not very hard to implement, but it was very ugly. It proved to be a lot easier to just check if there is NO conflict. If two programs don't conflict one of the following statements is true: a. Program A ends before program B starts b. Program A starts after program B ends. This check is a lot easier and if you negate it, it has the same result. This is the way normal conflicts are checked.

2. A case which is somewhat harder to check is if a normal program has a conflict with a periodic program. The easy way to do this would be to check for conflicts with every period of the periodic program in the same way as you would check for a conflict between two normal programs. This however is very inefficient and would even crash the program in case of a infinite periodic program. A better way to check for conflicts would be to first check which periods could give a conflict and then only run the check on these periods. The first period that could result in a conflict can be calculated in this way: x=the start time of the normal program. y=the end time of the first period of the periodic program. z=the period length of the periodic program.

$$\frac{x-y}{z} = the\ first\ period\ in\ which\ a\ conflict\ could\ occur.$$

The last period that could result in a conflict can be calculated in a similar way. x=the end time of the normal program. y=the start time of the first period of the periodic program. z=the period length of the periodic program.

$$\frac{x-y}{z} = the\ last\ period\ in\ which\ a\ conflict\ could\ occur.$$

This formula is explaind by the following timeline.

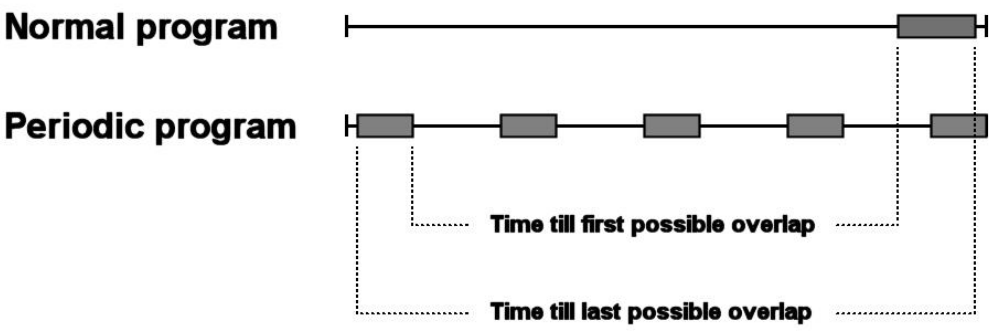

Figure 9.3: Timeline to explain conflict checking

3. The third case is to check for conflicts between two periodic programs. This case is even harder than the second case. Again you could just compare every instance of the first program with every instance of the second, but this would give the same problems as before. Again we will handle this more efficiently by defining the range in which a conflict could occur, this is somewhat harder then the in the second case since both programs are periodic. This is how we solved this problem: Program A has a repeating behaviour in period length A (and therefore also in every multiple of this length), Program B has a repeating behaviour in period length B (and therefore also in every multiple of this length). So if you look at the behaviour of the two programs together it is repeating in every period with a length which is a multiple of period length A en period length B. The smallest of these lengths is the "smallest common multiple" of A and B. So if there is a conflict between the two programs it should occur in a period of this length. In this period you can just check all the instances of one program for conflicts with the other program (in the same way as case 2).

In this way you can check for every possible conflict and thus you guarantee a feasible program.

9.4.3 Configuration files

To prevent the loss of data in case of a crash or a shutdown rivo stores the lists also as files on the hard drive. These files are written according to a grammar so they can be parsed again. This was a part of the implementation of the module that proved to take more time than expected. Designing the grammar

was quite easy (see appendix B), but to parse the files back to a linked list took more time than expected. It wasn't really hard to figure out what the parser should do, but to get it done in c was terrible, mainly because c handles strings "not really well" (understatement). To make the parser as reusable as possible the parsing goes in two steps:

1. The pre-parser removes all the comments and white space and puts the remaining strings in a linked list of "blocks". This part is reusable for all kinds of configuration files.

2. The final parser is to change the blocks into data the program can use (programs for instance).

Using this system of parsers it is possible to reload data stores when the system starts. To make sure that no user programs are lost RiVo synchronizes the stores with disk if they are changed and reloads them when the system starts. When the system starts RiVo removes all the programs which passed while the system was down (it also removes the instances of periodic programs which passed). In this way RiVo tries to execute the user commands as much as possible, even if the system has been down.

9.5 Problems

9.5.1 Periodic programs

The idea to have normal programs and periodic programs was nice in the users perspective, it could save the user a lot of time. Implementing it however was a rather different case. The problems started with the checking of conflicts between two periodic programs which is discussed in a previous part. The second problem was this: If the user wants to see the schedule it would be logical if he gets the normal programs and the instances of periodic programs, if he would want to remove a program however he wouldn't be able to remove all the programs, the instances of periodic programs are not removable by itself in our current implementation. We thought of two ways to solve this problem:

1. When the user asks for the schedule the planner could change the first instances of the periodic programs to normal programs so the user can operate on them like he would on normal programs (in fact they are normal programs now). This solution however has a couple of drawbacks. When the users removes a periodic program the parts which are stored as normal programs should be removed too. This would request a link between normal programs and periodic programs. Another problem is the fact that if the user would ask for a schedule far in the future all the instances of the periodic programs would go to memory and this would cost a LOT of memory.

2. The second solution is to make a single instance of a periodic program removable in this way. If you remove a instance of a periodic program you could split the original program in two parts around the instance to remove, this would be a very memory efficient way and it wouldn't require that much changes in the implementation. Unfortunately we hadn't enough time to do this, therefore the problem is not solved in our current release.

9.6 Testing

The testing of the module was done in the following way: All the functions were reviewed, and tested so that all the code was ran at least once (by testing the alternative of every if statement). This way proved at least that the code was working on itself. Testing for deadlocks was a lot harder however, these only occurred in special cases were the timing was just right. Since it is impossible to check every possible state of the program we could only test for this by running the program thoroughly. Also we tried to avoid complex deadlocks by keeping the synchronization in this module as easy as possible.

9.7 Planner

Though the planner works rather nice at this time there is a lot of space for improvement. The most important thing is the problem discussed above, the removing of instances of periodic programs. But there are more things. At this time all the configuration files are stored in our own file type with our functions. It would be a lot of work to add functions to sort the data of search through the data. Therefore it might

be a better solution to use an external database library to do these things. An other thing which was in the original design but which is not yet implemented is this: At first we wanted the program to be able to go though channel guides at its own, and record programs based on user profiles. Due to the lack of time and the absence of online channel guides we were not able to implement this feature yet. The design and the code however are based on the idea that this feature will be added so therefore it is possible to build in this feature without big changes in the code.

Chapter 10

The control interface module

10.1 Goal

CIF stands for 'Control InterFace'. That describes the purpose of CIF nicely: providing an interface for the world to control rivo.

10.2 Requirements

CIF module has the following requirements:

- accept client commands and parameters
- execute internal rivo functions
- report outcome of executions to the client
- any inactive client should be disconnected
- it should be possible to regularly send audio buffer information to the client

10.3 Design

The entire interface to rivo consists of 2 parts:

1. the control interface (cif, the part compiled into rivo)
2. the user interface (uif, the part which interacts directly with the user)

These two parts communicate over a standard tcp/ip-connection. Tcp/ip was an obvious choice due to the widespread use and support of tcp/ip. This 2-part network design has several advantages:

- the user interface can be implemented in any language supporting tcp/ip connections. (PHP, Java, Perl, C etc. etc.)
- the user interface can run stand alone; e.g. on a different operating system than rivo.
- it makes rivo always easy accessible.

With this design many different kinds of user interfaces are possible. Examples are CGI's running on a server, Java programs running locally and even using plain telnet as your user interface. This chapter will only discuss the control interface module (cif). The user interface module (uif) will be discussed in the next chapter.

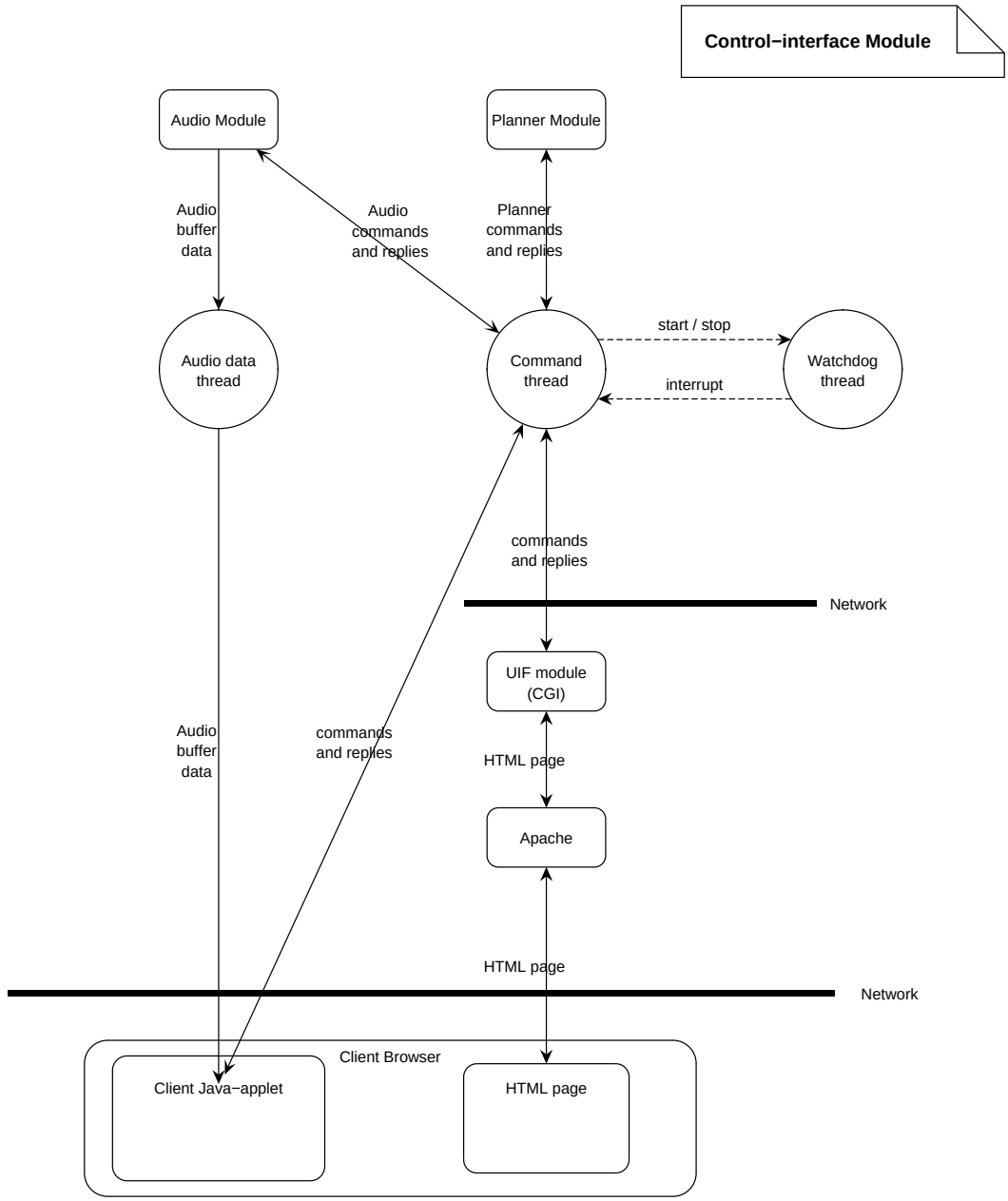

Figure 10.1: Interface design

10.3.1 Threading design

CIF has been designed as follows: there are 2 threads running continuously. We call them ’command thread’ and ’audio data thread’.

The command thread starts by waiting for a client to connect on its port. Once connected it will wait for client input. When a client has sent input, it will parse the input and executes the appropriate internal rivo functions. After that the outcome of the internal functions will be sent back to the client. Thus, the initiative of communication always lies with the client. The command thread will not suddenly start sending data to the client; only as a reply to a user action.

Since in the current design there is only one thread running to accept commands, only one client can be handled at a time. This causes a necessity to keep the port to be free as much as possible: you don’t want an inactive client to be connected forever and block the interface. This is why, once a client is connected, another temporary thread is started: the watchdog thread. This watchdog thread makes sure that a client

gets disconnected, if it has been inactive for a certain period of time.

The second continuously running thread, audio data thread, also starts by waiting for a client to connect on its port. Once connected this thread starts sending audio buffer information to the client. This thread does not accept any client input whatsoever. Here, the initiative lies with the audio data thread.

The original idea was to start the audio data thread only once a client had sent such a command to the command thread. Why keep this audio data thread running if you already got the command thread running continuously?

The continuously running audio data thread doesn't use processor time if it is listening on a port for a connection. So the continuously running thread doesn't spoil much more resources compared to the dynamically started audio data thread. The dynamically started audio data thread however introduces the need for extra commands in the command thread and thread-start/stop complexity. Also, clients that are interested in audio data have to be more complex in case of the dynamically started audio data thread (they first have to connect to the command thread, send the 'start audio data thread'-command and then connect to the audio data thread. These disadvantages caused us to go with the continuously running audio data thread.

10.4 Implementation

10.4.1 cif_init

The rivo main function launches the CIF module by calling cif_init with some parameters such as the command thread port, audio data thread port and timeout length. These are copied internally in CIF for further usages.

Next, CIF requests 2 sockets from the operating system. This is being done with the help of 'sockLib', a socket handling library developed by dhr. Schoute and others at the University of Twente. If the socket requests fail, cif_init aborts and returns an error value to the rivo main function.

Normally the socket requests succeed and CIF then starts the 2 threads. Both threads are accompanied by a shutdown mutex. The shutdown mutex makes sure that the thread is in safe code, when it is going to be stopped. As soon as the thread goes into code in which stopping would be unsafe, it locks the mutex. The thread unlocks the mutex when it leaves the unsafe code.

10.4.2 cif_close

When the rivo main function calls the cif_close function, the following procedure is executed for both threads: the close function tries to get a lock on the shutdown mutex. Once this lock is acquired, the thread has apparently released the mutex and thus is in a state in which it can safely be stopped.

After that a cancel signal is sent to the thread. The close function waits for the thread to die by joining the cancelled thread. Now we can be sure the thread has safely been closed.

Once both threads have been stopped by this procedure their sockets are returned and the shutdown mutexes destroyed.

10.4.3 The command thread

The command thread begins by setting its cancel reaction to asynchronous. Asynchronous cancelling would be unsafe, so we make the cancelling synchronous by using our own shutdown mutex as explained above. We do not use the provided synchronous cancelling because it works by completely enabling or disabling cancelling. This way, a cancel signal will get lost when the thread has disabled cancelling and thus the thread would keep on running. This is not what we want, so we use the shutdown mutex; a lock on this mutex never gets lost.

After handling cancel reaction, the command thread starts a loop in which it listens on the given port for a connection. This is also being done with help of 'sockLib'. If it can't start listening on the given port, the loop will be aborted. Another reason to abort this listen loop is the shutdown command from the user. When the listen loop has been left, the rivo main shutdown mutex is being unlocked, causing the rivo main to wake up and shutdown the whole program.

10.4.4 Client handling

Inside the listen loop client handling is being done by a separate handle client function. This function consists of the following loop:

- write a standard prompt to the client
- start watchdog thread
- read all client input
- stop watchdog thread (if it is still alive)
- parse input and execute it
- report execution outcome to the client
- end with ok- or error-message.

This loop continues until the client enters the abort command or the connection has been lost.

The prompt is very important for the client because once a prompt occurred, the client knows it has received all output from the previous command and can start sending a new command.

The watchdog thread only lives when the command thread is being blocked by the read system call (inside cif_read). If this read call takes longer than a given period of time, the watchdog thread wakes up and closes the connection. This forces the read system call to return and stop blocking command thread. The command thread detects an error and stops handling the client immediately.

While the client input is being parsed and executed, functions from the planner and audio module are called. It would be very unsafe to stop the command thread while it is executing a planner- or audio-function. This is why the shutdown mutex gets locked before parsing and executing and is released after.

The final ok- or error-message is for consistency. Every output from parsing and executing the client command ends with one of both. After this the prompt occurs. This way, the client can always check for an error by simply parsing the line before the prompt.

10.4.5 cif_read and cif_write

For easy and consistent communication with the client, we've made cif_read and cif_write. These functions wrap around the read and write system call.

The ordinary read system call requires a fixed size buffer to store its data in. However, the client input might exceed this buffer size. For example: when adding a new periodic program. The add command and all the data in the fields of the new periodic program need to be sent to the server. This command and its parameters could very well be longer than read expected. Data exceeding the buffer size will arrive at a next read system call (and then be interpreted as a new command).

The function cif_read has to prevent this. It does so by collecting client input, until an end-of-line character has been received. The collecting consists of calling the read system call, creating a new and bigger buffer if necessary, copying old data and newly read data into the new buffer and checking for the end-of-line character. If the end-of-line character doesn't appear the collecting continues. Eventually the entire client input buffer is being returned.

Cif_write has currently only debugging purposes but could later be used for same purposes as cif_read.

10.4.6 Command handling

After reading, the command thread parses the client input. Client input is always parsed as a command with zero or more parameters. The command can of course be unknown. The list of possible commands is given in appendix C.

The command is separated from the arguments with a mark character. This mark character also separates arguments from each other. A parameter however, could contain the mark character as an ordinary character (for example, the user just wants the mark character in the name field). In this case the mark character should be escaped.

This can't be done with the mark character itself because it would cause a symmetry problem: what would a string of 3 mark characters mean? It could be: escaped mark character followed by a real mark character or the other way around. Therefore we need another special character: the escape mark character. This character escapes the normal mark character. Now the symmetry problem is solved because the escape mark character isn't the same as the mark character and thus escaping can be distinguished from separation. The escape mark character can safely be escaped with itself. A single escape mark character escapes something; a double escape mark character is an escaped escape mark character.

The adding and removing of mark characters and escape mark characters is called stuffing and destuffing. Everything that cif needs to send to the client is being stuffed first and then packed together with mark characters between the parameters. Everything cif gets from the client will be broken into parameters and then destuffed. To save processor time stuffing and destuffing is only applied to parameters that represent strings such as name and description. For parameters representing integers it would be pointless (unless you pick a number as mark character).

The parsed input will be compared against all known commands until the corresponding command is found; if no command is found, the command thread will send a 'unknown command'-error to the client. If the corresponding command is found, the appropriate execution will follow. This consists of a simple syntax check on the parameters and some planner- or audio-function calls. The outcome will be sent back to the client.

10.4.7 The audio data thread

Cancelling in the audio data thread is handled exactly the same way as in the command thread. Next, the audio data thread starts a loop in which it listens for client connections (also the same as the command thread does). Once a client is connected, the audio data thread starts sending audio data on a regular interval. It gets this data by calling functions from the audio module. Just before calling these functions the shutdown mutex is lock again and released after the function calls returned; again to make cancelling this thread safe.

10.5 Problems

In the current cif design there are a few problems:

1. The currently used socket library doesn't have enough functionality. This problem manifested itself first when we wanted to print the ip-address of the client to the debug output. We expected the library to be capable of handling this but unfortunately it did not. The alternative is to search for a more advanced socket handling library and start using that.

2. Security checks are not present. The focus of this project was on functionality; therefore security has not been taken into account.

3. Only one client can be handled at a time. Since usually only one tuner is available per server, making rivo completely multi-user would not be a huge advantage. For this reason we decided to stay single-user and put more time in other aspects.

4. Currently the watchdog thread gets started and stopped a lot. It would probably be more efficient to start the thread once and let it switch between 2 states: 'counting down the timeout' and 'waiting to start count down'.

10.6 Testing

Testing of this module happened by using Linux telnet. We opened a telnet connection to the server and tried all commands with different kinds of parameters.

10.7 Future

One of the first things that will be fixed in this module, is security. Although this might make things a little more complicated for the user interface we believe it is a very important issue. We can achieve this

by calling a username/password verification function before the client handling function is called. More security can be achieved by checking the ip-address of the client against a given list of allowed ip-addresses (for example in a configuration file).

Although rivo will probably not get multi-user in the near future, handling multiple clients does have its advantages. Currently, clients are put in a queue if they connect while there is already a client being serviced. However, it could be that some client only wants to get informed. Read-only commands like that could safely be handled simultaneous.

This got us to the idea to split the command thread into 2 threads

1. an information providing thread (read-only commands)

2. an information accepting thread (write commands)

The threads would be running continuously and listening on separate ports. The first thread only provides information about rivo. For each connecting client it starts a new thread, so rivo can handle multiple clients with read-only-commands simultaneously. It doesn't need very strict security regulations since only read commands can be executed. If the connected user interface supports it, the client handling thread could also use push-technology (suddenly sending updated information to the client). The audio data thread can then be replaced by this information providing thread.

The second thread accepts the write commands. It has very strict security measures and is only capable of handling one client at a time.

The disadvantage of this separation is that the client side gets more complex: it has to decide which port to use for which commands. However more clients can be handled, security can be more precise and push technology can be used.

Chapter 11

The user interface module

11.1 Goal

Provide a nice user interface for rivo.

11.2 Requirements

- convert user actions into control interface commands
- send these commands to rivo
- give a comprehensible overview of the status of rivo

11.3 Design

Although many different user interfaces are possible, we decided to implement one design: a cgi/html-server combination. This design has one big advantage: it is very well supported on different systems: every computer with an Internet connection and a browser will be able to use it. The disadvantage is that HTML does not support push technology. So it is not possible to suddenly update the user interface because of a spontaneous action within rivo. (Of course there are workarounds for this problem but that would cause incompatibilities between different browsers.)

Concretely, the cgi/html-server combination comes down to the following procedure: the user points his browser to the web server containing the cgi-programs. The web server runs the cgi-program and that cgi-program interacts with rivo. The outcome of the conversation with rivo results in a HTML-page. The web server then sends this HTML-page back to the users browser. All user actions will follow the this route: browser sends information to the web server, passed on to the appropriate cgi-program, resulting in a HTML-page, which is then sent back to the user.

11.4 Implementation

To implement this design the following components are necessary:

1. a web server, able to run external programs
2. the cgi-programs

11.4.1 The web server

For the web server we picked Apache. This was the easiest choice, since Apache was already installed and running happily on our server. The only thing we had to do, was to add an entry to Apaches configuration file for our directory containing user interface files. This entry makes sure the directory is served with the correct permissions.

11.4.2 CGI-programming language

The cgi-programs could have been written in a many languages (like PHP, Perl, Java etc.) but we chose c again. This was for several reasons:

Speed: Other common cgi languages (PHP, Perl etc.) are often interpreted and thus slower.

Compatibility with web servers: Many web servers are by default able to execute compiled c programs. Executing PHP programs requires a PHP-module, which is not yet very standardized.

Code reuse: In the cgi-programs many problems from cif reoccur. C solutions for this, can be reused. This happened for problems like stuffing/de-stuffing, sending/receiving strings, breaking input into parameters, converting time strings to absolute seconds and back, etc.

Definition reuse: Some header files from cif could be used. For example the list of commands is defined only once: in the cif header file. The cgi-programs use these (and others) directly.

Of course there were also disadvantages of using c:

String manipulation: In c string manipulation has to be done very carefully. In languages like Perl and PHP this can be done much easier.

Less standard functions are available: Things like finding a specific line in a file needs more code in c than in Perl or PHP.

11.4.3 CGI-program internals

Any cgi-program begins by parsing the 'action' variable. This variable determines what must be done by the cgi. It is set by the browser, depending on the user action. If the action variable has not been set at all, the cgi reverts to showing a standard overview.

For example, when the user clicks on 'edit', the action variable will be set to 'edit'. The cgi-program detects this and thus generates the edit form with the correct input fields. When the editing is finished, the user clicks on 'done', which sets the action variable to something like 'edit_done'. This causes the cgi to parse all input fields and send edit commands to rivo.

The controlling cgi-programs like radio, wave player and buffer control are a little more complex. Before reacting on the action variable, they ask rivo for the current state. Depending on that state, the appropriate commands are send to the server. After sending commands, the complete rivo state is asked again and based on the state a HTML-page is generated . For example, when the action is 'start_recording' it first checks if there is a program running. If so, then that program should also be recorded. If there is currently no program running, a new program will be added, with the record variable set to 'yes'.

Before displaying output, any cgi-program will read a given HTML file. This HTML file contains plain HTML code and a special mark tag somewhere between HTML code. Everything before the mark tag will be read and send to the user. When the mark tag appears, it is replaced by the cgi output. When the cgi output is finished, the rest of the file is read and send to the user.

This enables us to separate dynamic html-code as much as possible from the static html code. The file containing the static html code can be changed in any way, without the cgi-program having to know about it. This gives great flexibility over website design. No need to recompile cgi's if you only wanted to change font size, background colour, screen layout etc. Only if you want to change table layout, the cgi-program should be adjusted.

11.4.4 Used libraries

The user sets some variables inside the browser by performing actions on the site. The web server receives the variables from the browser. To get these variables inside the cgi-program we've used 'libcgi'. This library provides standard functions to implement cgi-programs in c.

For exchanging information with rivo we've made our own library. This is called 'uif_cif_lib'. It provides a 'send_to_server' function, which accepts a string (the command that must be sent to rivo) and returns an array of strings (each line that rivo gave as reply). Everything else, such as opening the connection, writing, reading, etc. is handled by this lib. Calling 'send_to_server' a lot of times doesn't make it inefficient: the first time it opens a new connection to rivo; after that it remains logged in in rivo for as long as the cgi is running.

11.4.5 Configuration file

The send_to_server command has 2 more important parameters: hostname and port. These parameters are always provided by the cgi-program. In the current implementation each cgi-program reads a simple configuration file containing the hostname and port. These settings are then used. The advantage is that their value can be changed at runtime. Suddenly using another server or another port is now possible.

The configuration file contains one other entry: the applet port. The cgi-program that generates the page with the applet on it (buffer control) uses this parameter to direct the applet to the correct server and ports.

11.4.6 safe_html

Using HTML as design language in combination with the possibility of entering any data you like in the data fields, poses us for a small problem: what happens if the user, accidentally or not, enters HTML commands into a data field? That would cause corruptions in the resulting HTML-pages.

This problem is solved by checking newly entered data for HTML tags like <, > and ". These are replaced by text equivalents (like "&lt", "&gt" and "&quot"). Browsers will then show these text equivalents as the original characters but will not interpret them.

11.5 Problems

In the current implementation of the 'radio control' cgi-program we have chosen to use multiple buttons for submitting a single form. The form contains the selected channel; the button tells which action should be taken (start live, start recording, pause etc.) However it appeared that at least one browser is not capable of handling this kind of form (Pocket Internet Explorer, running on Windows CE on the Compaq iPAQ). This should of course be fixed.

Currently the user interface uses 2 frames: the upper frame to show information about channels, planning, periodic planning, audio files and preferences. The lower frame to show the radio control, wave player control and buffer control. Splitting up the screen allows to control rivo faster (for example, on one screen, you can control the radio while you are editing a channel) but tends to use more space. For systems with small screens this might become a problem. Another problem is that on a user action only one of both parts gets updated. This means you can get annoying inconsistencies like this: you removed a channel in the upper part of the screen but in the lower part, you can still select that channel in 'radio control'.

11.6 Testing

Testing has been done by trying out all the options. More systematic testing has been done in the testing period of the entire rivo system.

11.7 Future

The problems caused by using frames really need to be fixed. The best solution is to stop using frames anymore. This will make the user interface more straight forward and consistent.

As always with user interfaces: they can be nicer :) More pictures, different colors, better layout etc. It would be nice to start using Cascaded Style Sheets in the static HTML files. CSS allows for more control over the website design and thus nicer designs.

When the control interface (cif) gets stricter security measures, the user interfaces have to follow. This means for the current user interface that the library that takes care of the connection, probably has to be extended with username/password negotiation. However, it would be quite un-user-friendly to require entering username/password every time a cgi-program is launched. To solve such a problem, we could start using sessions or save the username/password in hidden input fields in the HTML-pages.

Chapter 12

The Buffer Control Applet

12.1 Goal

Provide the user with a way to control the live-buffer

12.2 Requirements

The applet has to work in all java-enabled browsers. The user should be able to wind, to rewind and to pause/resume in a convenient way.

12.3 Design

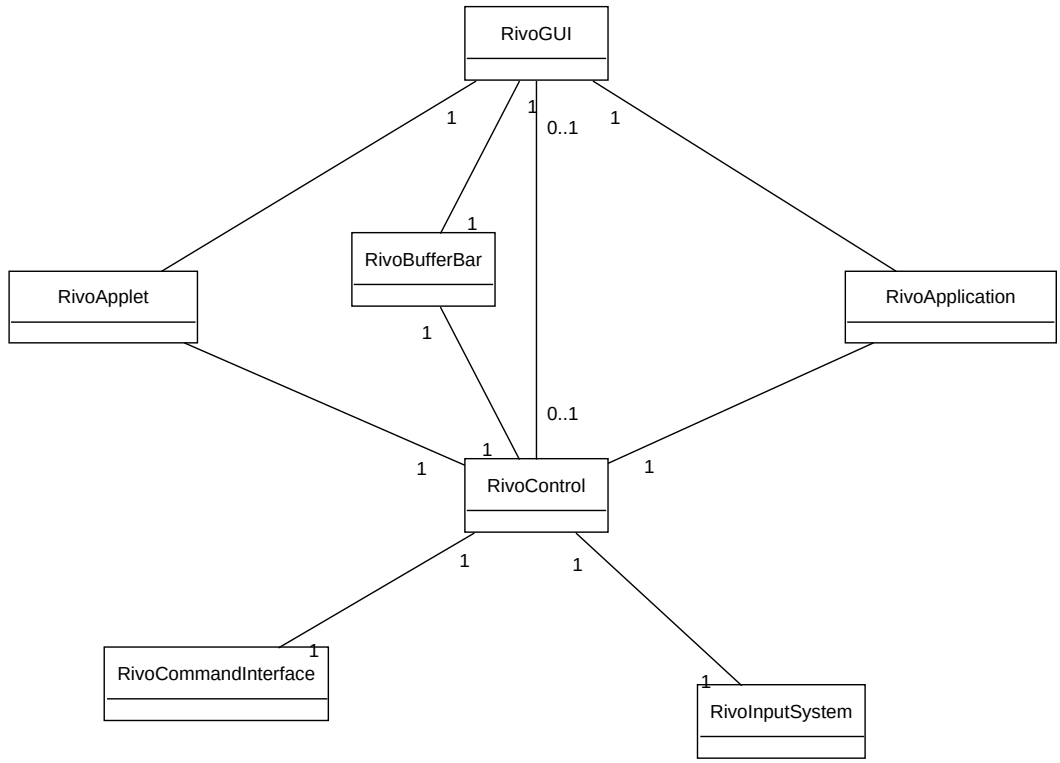

Figure 12.1: Applet design

There are 5 important classes in this design:

RivoBufferBar: This is an awt gui component that extends Canvas. It displays the current status of the buffer. It also handles mouseclicks.

RivoGUI: This is an awt gui component that extends Panel. It has 2 buttons: pause and resume. It also displays the RivoBufferBar.

RivoControl: This is a class that glues the gui component and the network code together. It passes commands from the RivoGUI to the RivoCommandInterface and it passes buffer updates from the RivoInputSystem to the RivoGUI.

RivoCommandInterface: This is a class that sends commands to the rivo server and returns the result.

RivoInputSystem: This is the class that continuously receives updates about the status of the live-buffer.

There are 2 other classes (RivoApplet and RivoApplication) that actually "wrap" the other components in an applet or in an application.

12.4 Implementation

The applet has to run in all java-enabled browsers. Because there are a lot of different browsers and a lot of different Java virtual machines, we had to look for a Java version that is supported by all common Java VMs. This was java 1.1 . We only used Classes/Methods from the java 1.1 API and with a little tweaking it finally worked in all java enable browsers.

12.5 Problems

There were some problems caused by incompatibilities between browsers. For example: it is in the java standard that when a browser leaves the web page, the destroy() method of the applet is called, but Netscape 4.7 for Linux doesn't.

Another problem is that when the applet is started when another applet is running, the second applet crashes. This is because the applet tries to connect to the applet-port of the server, but that port is already in use. The applet doesn't get a "Connection Refused" but is waits until it can connect.

12.6 Testing

We didn't bother to test the applet very thorough, we simply ran it to see if it worked or not.

12.7 Future

The design is very modularised, so if in the future the network-protocols change, only 1 or 2 classes have to be changed. It is also possible to make another gui with fancy animations or whatever people like.

Chapter 13

Testing

During implementation, of course we ran some tests to see if the code worked. However, testing in a more structured way is necessary. In this section we will describe how we tested the system as a whole and we will explain why we did it that way.

The individual modules were tested in an early phase by the implementer itself. Generally, this wasn't done in a very structured way, but the obvious bugs were found and fixed. This procedure tests the operation of the application as a whole.

We used the web-interface to test rivo and not the telnet-interface, because the telnet-interface is part of the internal implementation of the system and users usually won't use that interface at all. The interface that is meant to be used directly by humans is the web-interface and this needed to be tested, too. The web-interface relies completely on the telnet-interface and if there is a bug in the server we will see it anyway. Of course this makes bug searching / bug fixing a little harder, but in practice we haven't had any problems with that.

In some test cases we deliberately tried make the program crash. In these cases we opened 2 browser windows to the rivo interface, in order to call the cgi programs with incorrect or inconsistent parameters.

Before we ran the tests, we set up quite a lot of test cases. We did these tests and attached a label "passed" or "failed" to it. If the test failed, we wrote down what happened and if a fix was required or not. Later on we checked the problems again to see if they were fixed properly.

In the appendix the complete test cases can be found.

Chapter 14

Future options

In the future some things can be added/modified to the program. In this chapter these ideas are written down. Some of them are more realistic than others, but theoretically they are possible.

1. The configuration files can be integrated in the web interface. Then the user can control everything in the web interface.
2. More options in the configuration file (like bit rate).
3. Plugins for radio guides (the plugin system is already there).
4. User profiles. So that the system can plan for itself according to the profiles.
5. More output plugins, like ogg encoded out, encoded file writing and for other applications.
6. A variable user interface, adjusted to the right environment. A special interface for every circumstance, optimised for i.e. flash, VRML, text based browsers, handheld devices.
7. A user interface that has all the functions in one application, i.e. the mp3 player, buffer control, planning etc.
8. A live buffer that can process video data (i.e. from a TV card). So that tv can have the same convenience of rivo.
9. Support for multiple radio cards

Chapter 15

Review

An important final action in a project like this is to review the project to spot flaws in the product. Also the requirements are reviewed and we analyse which are not met and for what reason. In this chapter we will do these reviews.

15.1 Requirements review

15.1.1 Recording

The first requirement was about recording, the program should be able to record at a desired time on a desired channel. This requirement is met. The required functionality is implemented and there even are some extra, user friendly, features like the downloading of audio files and the possibility of adding so called "periodic programs" these are programs that repeat itself periodically which is useful if the user often wants to record a program at the same time.

15.1.2 Playing

The second requirement was about the playing of audio files. All the recordings should be offered to the user for playing. This requirement is not entirely met. The requirement document mentions rewinding and forwarding. These functions are not implemented because of the following reason: The audio module which would be responsible for this function uses a plugin system. Though this system makes the module very dynamic it also constrains the capabilities on a couple of fronts. To implement rewinding and forwarding a lot of functionality should be added to all the plugins and there was no time to do this. The other requirements like pause and resume however are fully implemented.

15.1.3 Listening to the radio

Pausing and resuming live radio broadcasts also was one of the demands. This requirement is perfectly fulfilled by the audio module, not only is it possible to pause and resume, it is also possible to fast forward and rewind through the buffered audio data.

15.1.4 User interface

The requirements about the user interface also are met for the most important part. It is a web interface so the user can access it from almost everywhere with his favourite browser. The parsing of radio guides however is not implemented for a couple of reasons:

1. We were not able to find parsable radio guides, this made it almost impossible to implement and therefore the requirement was not realisable. We tried to contact the NOS about this point, but the only answer we got was that they were legally not allowed to give that information to 3th parties (which seems very odd since they also put it on the internet in a non parsable way).

2. We ran out of time and had to make concessions. To compensate for the absence of the feature we did the following thing: we put links to the websites of the channels in the user interface so the user himself can look for radio guides.

15.1.5 User profiles

There is not very much to say about this requirement, the only part that is met is "keep it simple", we sure did. Since there were no online radio guides to parse and we ran out of time this requirement isn't implemented at all. The planner module however is constructed with the idea that this feature will be added later so adding this won't require lost of changes.

15.2 Hardware requirements

These requirements are mainly related to the mp3 encoding, because that part requires much cpu time:
We tested rivo on a p2 233 MHz with 64 mb memory. With the best quality (44 kHz,2 channels,16 bits, 128 kbps) the cpu usage was 80% and a load average of 1.00. With these settings the audio doesn't malfunction. With 2 channels,16 bits, 128 kbps and 22 kHz, the cpu usage was 30% and with a load average of 0.4. These readings make clear that a p266 with 64 mb memory can handle the encoding good. The rest of the program (without the encoding of mp3) uses less than 1% of the cpu and about 2% of the memory. That means rivo (without streaming) can run on a far less capable computer.

15.3 Irregular period lengths

Although periodic planning is a nice feature to have, it presented us with a small problem: is it possible to use irregular period lengths? For example, months or years; these can not be converted to seconds since it differs from month to month and year to year. This is a shortcoming of our design, however it is very unlikely to be useful. That is because if a radio broadcast would be monthly, each time it would be broadcasted on a different day of the week. Planned broadcastings like this are highly unlikely.

15.4 PDA rivo control

Listening to rivo and controlling it via a pda was an extra challenge. We tried to make the pda work with our program, but we had 2 problems:

1. Some generated HTML-pages are not compatible with Pocket Internet Explorer.
2. The mp3 streaming didn't work, although the streaming of static files with icecast did work.

These problems weren't solved, since time forced us to focus on more important issues. However, the encountered problems can be solved in the near future.

15.5 Safety

Since this product has a web interface and therefore is accessible all over the world safety is a important point of consideration. Although it has no built in security measures we tried to keep this product as safe as possible. We avoided common vulnerabilities like buffer overflows by using strncpy instead of strcpy and prevented pointers from pointing to strange points in memory by using calloc instead of malloc. But still a good firewall to restrict the use of this program to trusted clients is advised. (disclaimer blablabla).

15.6 Stability

Unlike some commercial corporations we won't name here, we think that it is important that programs don't crash all the time. To accomplish this we tried to keep the synchronisation as simple as possible to avoid deadlocks and we ran tests on the program for long periods of time to find all bugs. Finally we think

that our program is rather stable when used in normal ways, there are however a couple of ways to tease the program into bad behaviour. The first way is by using the two possible ways of control at the same time. In the telnet interface you can add programs that have a somewhat negative impact on the web interfaces performance (adding a program named "</table></html>" will most certainly give a undesired effect). The second way is by manually editing the configfiles with faulty data, this also will certainly give unexpected results. Both cases however will not happen with normal use, so only if someone wants the program to do strange things this will happen. And even then the program only does what the user wants (meaning: it acts strange), so it is very user friendly.

Overall we think that, although some requirements are not met, our product is as good as we could make it in this time. If we would do such a project again we would certainly be able to create a better product, but that is because we learned a great deal during this project.

Chapter 16

Teamwork

16.1 Our method

Our method of teamwork is, to thoroughly discuss all the problems/difficulties and to have everything at one point. This is because then we are at one line with our thoughts. This method is found back in several decisions:

1. There is one place where we keep our source code, documents and other project related stuff (i.e. links). To make this happen, we chose for CVS.
2. We have documents and functions for the debug standard, code layout standard, document layout standard, makefile standard, plugin standard and a minutes standard. This is to keep everything uniform.

Furthermore we scheduled everything (with some space for delay) from the beginning, to keep a tight schedule. In this way we didn't lose control of the situation. We used milestones like feature freeze and code freeze to realise this. The planning can be viewed at D. We kept minutes for every appointment we had with each other and our accompanists. These came in handy as to support our memory.

16.2 Software tools

We used several software tools for testing, constructing our project.

1. For testing the web interface we used: konqueror, netscape, mozilla, internet explorer, w3m and links.
2. For testing the streamer we used: mpg123, xmms, sonique and winamp
3. For constructing our project we used: gcc, make, cvs, lame, icecast, apache and several libraries (i.e. libshout and stdlib).

16.3 Team setup

In the beginning of the project we divided the responsibilities for a clear division. The following modules were given to the following people:

1. (aud) Jeroen: Module with the name audio, responsible for:
   (a) Reading the radio card.
   (b) Buffering.
   (c) Delivering the audio data at the right places.
2. (pln) Ardjan: Module with the name planner, responsible for:
   (a) Managing the radio card.

   (b) Collecting and processing radio-information.
   (c) Managing profiles.
   (d) Planning, starting and stopping of recordings.

3. (str) Barry: Module with the name streamer, responsible for:
   (a) On-the-fly compression of audio data.
   (b) Delivering the audio data to clients over a network.

4. (cif) Matthijs: Module with the name control-interface, responsible for:
   (a) The interaction with the user over a network.
   (b) Interpreting user commands and sending the interpreted commands to other modules.
   (c) flexible UI.

There was also an organizational division. The following tasks were assigned to the named people:

1. Barry Nijenhuis: secretary.
2. Matthijs van der Kooij: chairman.
3. Ardjan Zwartjes: document manager (CVS, user-accounts) and document standards.
4. Jeroen Soesbergen: coding standards (code, remarks, debug-output, etc.).

These divisions were made in good deliberation with each other, so that everyone did his job with reasonable pleasure. We had a meeting every Monday, Tuesday and Thursday with each other to keep close contact. Some weeks we had more meetings, because there was more to discuss, but the basic meetings were every week. Of almost every meeting we have minutes, because if we didn't used them, we were bound to forget things. We also had a weekly meeting with our accompanists (Jansen and Scholten) to discuss our progress, problems and decisions. Every member of the group kept a log to write down idea's, bugs, etc. This was to recollect everything that had been done between the meetings.

16.4 Conclusion

The teamwork in our group went well. We had regular discussions about al kinds of things, but in the end we always came to an agreement. The mood was always serious, but also relaxed. This was because we know each other well.

We learned that a positive attitude is the best way to cope with setbacks. Even when there seems to be no progression.

Appendix A

Test report

Appendix B

Configuration file grammar

Appendix C

Cif protocol

Appendix D

Planning

Appendix E

Installation guide