Liquidsoap Documentation

David Baelde, Romain Beauxis, Samuel Mimram

Jul 02, 2020

Contents:

CHAPTER 1

Liquidsoap

Liquidsoap is a powerful tool for building complex audio and video stream generators, typically targetting internet radios and webtvs. It consists of a simple script language, which has a first-class notion of source (basically a *stream*) and provides elementary source constructors and source compositions from which you can build the stream generator you want. This design makes liquidsoap flexible and easily extensible.

We believe that liquidsoap is easy to use. For basic purposes, the scripts consist of the definition of a tree of sources. You will quickly learn how natural it is to use liquidsoap in such cases. The good thing is that when you will want to make your stream more complex, you will be able to stay in the same framework and keep a maintainable configuration. Of course, using some complex features might require a deeper understanding of the concepts of source and request and of our scripting language.

We discuss below what liquidsoap is and what it is not. If you already know that and want to get started with liquidsoap, just jump to the documentation index. It guides you through these pages, starting with the quickstart tour.

Liquidsoap is an open-source sofware from the Savonet project.

1.1 Features

Here are a few things you can easily achieve using Liquidsoap:

- Playing from files, playlists, directories or script playlists (plays the file chosen by an external program).
- Transparent remote file access; easy addition of file resolution protocols.
- Scheduling of many sources, depending on time, priorities, quotas, etc.
- Mixing sources on top others.
- Queuing of user requests; editable queues.
- Sound processing: compression, normalization, echo, soundtouch, etc.
- Speech and sound synthesis.
- Metadata rewriting and insertion.

- Arbitrary transitions: cross-fade, jingle insertion, custom, etc. The behaviour of the transition can be programmed to depend on metadata and average volume.
- Input of other streams: useful for switching to a live show. Liquidsoap can relay an HTTP stream but also host it.
- Blank detection.
- Definable event handlers on new tracks, new metadata and excessive blank.
- Multiple outputs in the same instance: you can have several quality settings, use several media or even broadcast several contents from the same instance.
- Output to Icecast/Peercast/Shoutcast (MP3/Ogg) or a local file (WAV/MP3/Ogg/AAC).
- Input/output via Jack, ALSA, OSS and PortAudio. Output via `libao`.
- Interactive control of many operators via Telnet or UNIX domain socket, and indirectly using scripts, graphical/web/IRC interfaces.
- Video streams generation.

If you need something else, it's highly possible that you can have it – at least by programming new sources/operators. Send us a mail, we'll be happy to discuss these questions.

1.2 Non-Features

Liquidsoap is a flexible tool for processing audio and video streams, that's all. We have used it for several internet radio projects, and we know that this flexibility is useful. However, an internet radio usually requires more than just an audio stream, and the other components cannot easily be built from basic primitives as we do in liquidsoap for streams. We don't have any magic solution for these, although we sometimes have some nice tools which could be adapted to various uses.

Liquidsoap itself doesn't have a nice GUI or any graphical programming environment. You'll have to write the script by hand, and the only possible interaction with a running liquidsoap is the telnet server. However, we have modules for various languages (OCaml, Ruby, Python, Perl) providing high-level communication with liquidsoap. And there are at least two graphical applications using the Python module for controlling a running liquidsoap: `liguidsoap` (distributed with liquidsoap) and `soapbox` (another project of Savonet).

Liquidsoap doesn't do any database or website stuff. It won't index your audio files, it won't allow your users to score songs on the web, etc. However, liquidsoap makes the interfacing with other tools easy, since it can call an external application (reading from the database) to get audio tracks, another one (updating last-played information) to notify that some file has been successfully played. The simplest example of this is bubble, RadioPi also has a more complex system of its own along these lines.

CHAPTER 2

Documentation index

How to use: Start with the quickstart and make sure you learn how to find help. Then it's as you like: go for another *general tutorial*, or a *specific example*, pick a *basic notion*, or some examples from the cookbook. If you've understood all you need, just browse the reference and compose your dream stream.

If you downloaded a source tarball of liquidsoap, you may first read the build instructions.

If you are looking for a way to build a distribution-independant bundle of liquidsoap, you may want to read the custom path page.

2.1 General tutorials

- Quickstart: where anyone should start.
- Complete case analysis: an example that is not a toy.
- Advanced: overview of more advanced features for serious usage.
- How to find help about operators, settings, server commands, etc.
- Cookbook: contains lots of idiomatic examples.
- Frequently Asked Questions, Troubleshooting

2.2 Reference

- API reference: All the builtin functions of liquidsoap.
- Protocols: List of protocols supported by liquidsoap.
- Settings: The list of available settings for liquidsoap.
- Script language: A more detailed presentation.
- Encoding formats: The available formats for encoding outputs.

- Playlist parsers: Supported playlist formats.
- JSON import/export: Importing and exporting language values in JSON.
- LADSPA plugins: Using LADSPA plugins.

2.3 Core

- Basic concepts: sources, clocks and requests.
- Stream contents: what kind of streams are supported, and how.
- Script loading: load several scripts, learn about the script library.
- Execution phases

2.4 Specific tutorials

- Blank detection
- Customize metadata
- Seek and cue support: seek and set cue-in and cue-out points in sources.
- External decoders: use an external program for decoding audio files.
- External streams: use an external program for streaming audio data.
- External encoders: use an external audio encoding program.
- HLS ouptut: output your stream as HTTP Live Stream.
- HTTP input: relay external streams.
- Harbor input: receive streams from icecast and shoutcast source clients.
- Interaction with the Harbor: interact with a running Liquidsoap using the Harbor server.
- Interaction with the server interact with a running Liquidsoap instance using the telnet server.
- ICY metadata update: manipulate and configure metadata update in Icecast.
- Normalization and replay gain: normalize audio data.
- Requests-based sources: create advanced sources using requests.
- Shoutcast output: output to shoutcast.
- Dynamic source creation: dynamically create sources using server requests.
- Smart crossfading: define custom crossfade transitions.
- Using in production: integrate liquidsoap scripts in a production environment.
- Liquid Flows: add your radio to the webpage of proud users.
- Videos streams: why restrict yourself to sound only?

2.5 User scripts

- Bubble: a simple example of a database interface as a custom protocol.
- Geekradio
- RadioPi
- Frequence3
- Listen to Radio Nova
- Video with a single static image
- Split a CUE sheet

2.6 Code snippets

- Code example index

2.7 Behind the curtains

- Some presentations and publications explaining the theory underlying Liquidsoap
- OCaml libraries used in Liquidsoap, that can be reused in other projects
- Documentation of some internals of Liquidsoap
- Documentation for previous versions

CHAPTER 3

Advanced techniques

We shall now see two things that are essential to using liquidsoap fully: the server which allows you to control a running instance of liquidsoap, and the usage of the `init.d/liquidsoap` service script for running your radios in production in a clean and convenient way.

3.1 Protocols

Protocols in liquidsoap are used to resolve requests URIs. The syntax is: `protocol:arguments`, for instance: `http://www.example.com`, `say:Something to say` etc.

Most protocols are written using the script language. You can look at the file `protocols.liq` for a list of them.

In particular, the `process:` protocol can use an external command to prepare resolve a request. Here's an example using the AWS command-line to download a file from S3:

```
def s3_protocol(~rlog,~maxtime,arg) =
  extname = file.extension(leading_dot=false,dir_sep="/",arg)
  [process_uri(extname=extname,"aws s3 cp s3:#{arg} $(output)")]
end
add_protocol("s3",s3_protocol,doc="Fetch files from s3 using the AWS CLI",
             syntax="s3://uri")
```

Each protocol needs to register a handler, here the `s3_protocol` function. This function takes the protocol arguments and returns a list of new requests or files. Liquidsoap will then call this function, collect the returned list and keep resolving requests from the list until it finds a suitable file.

This makes it possible to create your own custom resolution chain, including for instance cue-points. Here's an example:

```
def cue_protocol(~rlog,~maxtime,arg) =
  [process_uri(extname="wav",uri=uri,"ffmpeg -y -i $(input) -af -ss 10 -t 30 $(output)
↪")]
end
add_protocol("cue_cut",cue_protocol)
```

This protocol returns 30s of data from the input file, stating at the 10s mark.

Likewise, you can apply a normalization program:

```
def normalization_protocol(~rlog,~maxtime,arg) =
  # "normalize" command here is just an example..
  [process_uri(extname="wav",uri=arg,"normalize $(inputi)")]
end
add_protocol("normalize",normalization_protoco)
```

Now, you can push requests of the form:

```
normalize:cue_cut:http://www.server.com/file.mp3
```

and the file will be cut and normalized before being played by liquidsoap.

When defining custom protocols, you should pay attention to two variables:

- `rlog` is the logging function. Messages passed to this function will be registered with the request and can be used to debug any issue
- `maxtime` is the maximun time (in UNIX epoch) that the requests should run. After that time, it should return and be considered timed out. You may want to read from `protocols.liq` to see how to enforce this when calling external processes.

3.2 Interaction with the server

Liquidsoap starts with one or several scripts as its configuration, and then streams forever if everything goes well. Once started, you can still interact with it by means of the *server*. The server allows you to run commands. Some are general and always available, some belong to a specific operator. For example the `request.queue()` instances register commands to enqueue new requests, the outputs register commands to start or stop the outputting, display the last ten metadata chunks, etc.

The protocol of the server is a simple human-readable one. Currently it does not have any kind of authentication and permissions. It is currently available via two media: TCP and Unix sockets. The TCP socket provides a simple telnet-like interface, available only on the local host by default. The Unix socket interface (*cf.* the `server.socket` setting) is through some sort of virtual file. This is more constraining, which allows one to restrict the use of the socket to some priviledged users.

You can find more details on how to configure the server in the documentation of the settings key `server`, in particular `server.telnet` for the TCP interface and `server.socket` for the Unix interface. Liquidsoap also embeds some documentation about the available server commands.

Now, we shall simply enable the Telnet interface to the server, by setting `set("server.telnet",true)` or simply passing the `-t` option on the command-line. In a complete case analysis we set up a `request.queue()` instance to play user requests. It had the identifier `"queue"`. We are now going to interact via the server to push requests into that queue:

```
dbaelde@selassie:~$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
request.push /path/to/some/file.ogg
5
END
metadata 5
[...]
```

(continues on next page)

(continued from previous page)

```
END
request.push http://remote/audio.ogg
6
END
trace 6
[...see if the download started/succeeded...]
END
exit
```

Of course, the server isn't very user-friendly. But it is easy to write scripts to interact with Liquidsoap in that way, to implement a website or an IRC interface to your radio. However, this sort of tool is often bound to a specific usage, so we have not released any of ours. Feel free to ask the community about code that you could re-use.

3.3 Using in production

The full installation of liquidsoap will typically install `/etc/liquidsoap`, `/etc/init.d/liquidsoap` and `/var/log/liquidsoap`. All these are meant for a particular usage of liquidsoap when running a stable radio.

Your production `.liq` files should go in `/etc/liquidsoap`. You'll then start/stop them using the init script, *e.g.* `/etc/init.d/liquidsoap start`. Your scripts don't need to have the `#!` line, and liquidsoap will automatically be ran on daemon mode (`-d` option) for them.

You should not override the `log.file.path` setting because a logrotate configuration is also installed so that log files in the standard directory are truncated and compressed if they grow too big.

It is not very convenient to detect errors when using the init script. We advise users to check their scripts after modification (use `liquidsoap --check /etc/liquidsoap/script.liq`) before effectively restarting the daemon.

CHAPTER 4

Blank detection

Liquidsoap has three operators for dealing with blanks.

On GeekRadio, we play many files, some of which include bonus tracks, which means that they end with a very long blank and then a little extra music. It's annoying to get that on air. The `skip_blank` operator skips the current track when a too long blank is detected, which avoids that. The typical usage is simple:

```
# Wrap it with a blank skipper
source = skip_blank(source)
```

At RadioPi they have another problem: sometimes they have technical problems, and while they think they are doing a live show, they're making noise only in the studio, while only blank is on air; sometimes, the staff has so much fun (or is it something else ?) doing live shows that they leave at the end of the show without thinking to turn off the live, and the listeners get some silence again. To avoid that problem we made the `strip_blank` operators which hides the stream when it's too blank (i.e. declare it as unavailable), which perfectly suits the typical setup used for live shows:

```
interlude = single("/path/to/sorryfortheblank.ogg")
# After 5 sec of blank the microphone stream is ignored,
# which causes the stream to fallback to interlude.
# As soon as noise comes back to the microphone the stream comes
# back to the live -- thanks to track_sensitive=false.
stream = fallback(track_sensitive=false,
                      [ strip_blank(max_blank=5.,live) , interlude ])

# Put that stream to a local file
output.file(%vorbis, "/tmp/hop.ogg", stream)
```

If you don't get the difference between these two operators, you should learn more about liquidsoap's notion of source.

Finally, if you need to do some custom action when there's too much blank, we have `on_blank`:

```
def handler()
  system("/path/to/your/script to do whatever you want")
end
source = on_blank(handler,source)
```

CHAPTER 5

Bubble

Bubble is a simple program which scans your audio files and stores their metadata in a SQLite database. It can rewrite paths into URI so that you can index remote files mounted locally and rewrite the local path into the general URI before storing it in the database. For example if you mount your Samba workground in `/mnt/samba/workgroup` using `fusesmb`, you'll ask bubble to rewrite `/mnt/samba/workgroup` into `smb://`.

Bubble has been designed to be interfaced with liquidsoap to provide a protocol for selecting files by queries on metadata. URI rewriting makes it possible to query from another machine than the one where the indexer runs, and also makes sure that the file will appear as a remote one to liquidsoap, so that it will be fully downloaded to a safe local place before being played.

To add the bubble protocol to liquidsoap, we use the following code:

```
bubble = "/home/dbaelde/savonet/bubble/src/bubble-query " ^
         "-d /var/local/cache/bubble/bubble.sql "
add_protocol(
  "bubble",
  fun (arg,delay) -> get_process_lines(bubble^quote(arg)))
```

You could then have an IRC bot which accepts queries like play ‘‘Alabama song" and transforms it into the URI `bubble:title="Alabama song"` before queueing it in a liquidsoap instance. The bubble protocol in liquidsoap will call the `bubble-query` script which will translate the query from Bubble to SQLite and return a list of ten random matches, which liquidsoap will try.

Although it has been used for months as distributed on our old SVN repository, bubble is mostly a proof-of-concept tool. It is very concise and can be tailored to custom needs.

CHAPTER 6

Liquidsoap 1.4.2

6.1 Forewords

Installing liquidsoap can be a difficult task. The software relies on a up-to date `OCaml` compiler, as well as a bunch of `OCaml` modules and, for most of them, corresponding C library dependencies.

Our recommended way of installing liquidsoap is via opam. `opam` can take care of install the correct `OCaml` compiler, optional and required dependencies as well as system-specific package dependencies.

The `opam` method is described in details in the documentation. We recommend that any interested user head over to this link to install the software via `opam`.

The following of this document describes how to install the software via its `configure` script and is intended either for system administrators or package maintainers.

6.2 Dependencies

Below is a list of dependencies, mostly OCaml libraries. Optional libraries provide extra features. They need to be detected by the `configure` script.

Most of the libraries are developed by the Savonet project and, in addition to being available through traditional distribution channels, are bundled in the liquidsoap-<version>-full.tar.bz2 tarballs for easier builds.

Libraries not developed by Savonet are:

- camlimages
- camomile
- gd4o
- ocaml-pcre
- ocaml-magic
- ocaml-sdl

- yojson

6.2.1 Mandatory dependencies:

| Dependency | Version | | ——————— | ——— | | OCaml compiler | >= 4.08.0 | | ocaml-dtools | >= 0.4.0 | | ocaml-duppy | >= 0.6.0 | | ocaml-mm | >= 0.5.0 | | ocaml-pcre | | | menhir | | | sedlex | |

6.2.2 Recommended dependencies:

| Dependency | Version | Functionality | | ——————- | ——— | ——————————— | | camomile | >=1.0.0 | Charset recoding in metadata | | ocaml-samplerate | >=0.1.1 | Libsamplerate audio conversion |

6.2.3 Optional dependencies:

| Dependency | Version | Functionality | | ——————- | ——- | ——————————————— | | camlimages | >=4.0.0 | Image decoding | | gd4o | | Video.add_text() on servers without X | | ocaml-alsa | >=0.2.1 | ALSA I/O | | ocaml-ao | >=0.2.0 | Output via libao | | ocaml-bjack | >=0.1.3 | Jack support | | ocaml-cry | >=0.6.0 | Sending to Shoutcast & Icecast | | ocaml-dssi | >=0.1.1 | DSSI sound synthesis | | ocaml-faad | >=0.4.0 | AAC stream decoding | | ocaml-fdkaac | >=0.3.1 | AAC(+) encoding | | ocaml-ffmpeg | >=0.2.0 | Video conversion using the ffmpeg library | | ocaml-flac | >=0.1.5 | Flac and Ogg/Flac codec | | ocaml-frei0r | >=0.1.0 | Frei0r plugins | | ocaml-gavl | >=0.1.4 | Video conversion using the gavl library | | ocaml-gstreamer | >=0.3.0 | GStreamer input, output and encoding/decoding | | ocaml-inotify | >=1.0 | Reloading playlists when changed | | ocaml-ladspa | >=0.1.4 | LADSPA plugins | | ocaml-lame | >=0.3.2 | MP3 encoding | | ocaml-lastfm | >=0.3.0 | Lastfm scrobbling | | ocaml-lo | >=0.1.0 | OSC (Open Sound Control) support | | ocaml-mad | >=0.4.4 | MP3 decoding | | ocaml-magic | >=0.6 | File type detection | | ocaml-ogg | >=0.5.0 | Ogg codecs | | ocaml-opus | >=0.1.1 | Ogg/Opus codec | | ocaml-portaudio | >=0.2.0 | Portaudio I/O | | ocaml-pulseaudio | >=0.1.2 | PulseAudio I/O | | ocaml-sdl | | Display, font & image support | | ocaml-shine | >=0.2.0 | Fixed-point MP3 encoding | | ocaml-soundtouch | >=0.1.7 | Libsoundtouch's audio effects | | ocaml-speex | >=0.2.1 | Ogg/Speex codec | | ocaml-ssl | >=0.5.2 | SSL/https support | | ocaml-taglib | >=0.3.0 | MP3ID3 metadata access | | ocaml-theora | >=0.3.1 | Ogg/Theora codec | | ocaml-vorbis | >=0.7.0 | Ogg/Vorbis codec | | ocaml-xmlplaylist | >=0.1.3 | XML-based playlist formats | | osx-secure-transport| SSL/https support via OSX's SecureTransport | | yojson | | Parsing JSON data (of_json function) |

6.2.4 Runtime optional dependencies:

| Dependency | Functionality | | ——————- | ——————————————- | | awscli | `s3://` and `polly://` protocol support | | curl | `http`/`https`/`ftp` protocol support | | ffmpeg | external I/O, `replay_gain` level computation, .. | | youtube-dl | youtube video and playlist support |

6.3 Installing via configure

The build processus starts with by invoking the `configure` script:

```
% ./configure
```

If you want a complete installation of liquidsoap, enabling a production use of liquidsoap as a daemon, you should pass `--with-user=<login>` and `--with-group=<group>` options to indicate which user/group you have created for liquidsoap.

Then, build the software:

```
% make
```

You can also generate the documentation for liquidsoap:

```
% make doc
```

It will generate the HTML documentation, including a version of the scripting API reference corresponding to your configuration.

Then, you may proceed to the installation. You may need to be root for that.

```
% make install
```

This will not install files such as `/var/log/liquidsoap` unless you have provided a user/group under which liquidsoap should be ran. This behavior can be overridden by passing `INSTALL_DAEMON="yes"` (useful for preparing binary packages).

If you need to run liquidsoap as a daemon, you can then have a look at liquidsoap-daemon.

CHAPTER 7

Clocks

In the quickstart and in the introduction to liquidsoap sources, we have described a simple world in which sources communicate with each other, creating and transforming data that composes multimedia streams. In this simple view, all sources produce data at the same rate, animated by a single clock: at every cycle of the clock, a fixed amount of data is produced.

While this simple picture is useful to get a fair idea of what's going on in liquidsoap, the full picture is more complex: in fact, a streaming system might involve *multiple clocks*, or in other words several time flows.

It is only in very particular cases that liquidsoap scripts need to mention clocks explicitly. Otherwise, you won't even notice how many clocks are involved in your setup: indeed, liquidsoap can figure out the clocks by itself, much like it infers types. Nevertheless, there will sometimes be cases where your script cannot be assigned clocks in a correct way, in which case liquidsoap will complain. For that reason, every user should eventually get a minimum understanding of clocks.

In the following, we first describe why we need clocks. Then we go through the possible errors that any user might encounter regarding clocks. Finally, we describe how to explicitly use clocks, and show a few striking examples of what can be achieved that way.

7.1 Why multiple clocks

The first reason is **external** to liquidsoap: there is simply not a unique notion of time in the real world. Your computer has an internal clock which indicates a slightly different time than your watch or another computer's clock. Moreover, when communicating with a remote computer, network latency causes extra time distortions. Even within a single computer there are several clocks: notably, each soundcard has its own clock, which will tick at a slightly different rate than the main clock of the computer. Since liquidsoap communicates with soundcards and remote computers, it has to take those mismatches into account.

There are also some reasons that are purely **internal** to liquidsoap: in order to produce a stream at a given speed, a source might need to obtain data from another source at a different rate. This is obvious for an operator that speeds up or slows down audio (`stretch`). But it also holds more subtly for `cross`, `cross` as well as the derived operators: during the lapse of time where the operator combines data from an end of track with the beginning of the other other,

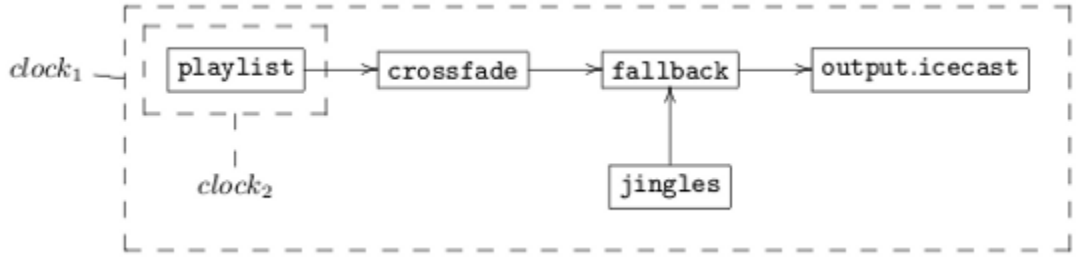
the crossing operator needs twice as much stream data. After ten tracks, with a crossing duration of six seconds, one more minute will have passed for the source compared to the time of the crossing operator.

In order to avoid inconsistencies caused by time differences, while maintaining a simple and efficient execution model for its sources, liquidsoap works under the restriction that one source belongs to a unique clock, fixed once for all when the source is created.

The graph representation of streaming systems can be adapted into a good representation of what clocks mean. One simply needs to add boxes representing clocks: a source can belong to only one box, and all sources of a box produce streams at the same rate. For example,

```
output.icecast(fallback([crossfade(playlist(...)),jingles]))
```

yields the following graph:



Graph representation with clocksHere, clock_2 was created specifically for the crossfading operator; the rate of that clock is controlled by that operator, which can hence accelerate it around track changes without any risk of inconsistency. The other clock is simply a wallclock, so that the main stream is produced following the ‘‘real” time rate.

7.2 Error messages

Most of the time you won’t have to do anything special about clocks: operators that have special requirements regarding clocks will do what’s necessary themselves, and liquidsoap will check that everything is fine. But if the check fails, you’ll need to understand the error, which is what this section is for.

7.2.1 Disjoint clocks

On the following example, liquidsoap will issue the fatal error `a source cannot belong to two clocks`:

```
s = playlist("~/media/audio")
output.alsa(s) # perhaps for monitoring
output.icecast(mount="radio.ogg",%vorbis,crossfade(s))
```

Here, the source `s` is first assigned the ALSA clock, because it is tied to an ALSA output. Then, we attempt to build a `crossfade` over `s`. But this operator requires its source to belong to a dedicated internal clock (because crossfading requires control over the flow of the of the source, to accelerate it around track changes). The error expresses this conflict: `s` must belong at the same time to the ALSA clock and `crossfade`’s clock.

7.2.2 Nested clocks

On the following example, liquidsoap will issue the fatal error `cannot unify two nested clocks`:

```
jingles = playlist("jingles.lst")
music = rotate([1,10],[jingles,playlist("remote.lst")])
safe = rotate([1,10],[jingles,single("local.ogg")])
q = fallback([crossfade(music),safe])
```

Let's see what happened. The `rotate` operator, like most operators, operates within a single clock, which means that `jingles` and our two `playlist` instances must belong to the same clock. Similarly, `music` and `safe` must belong to that same clock. When we applied crossfading to `music`, the `crossfade` operator created its own internal clock, call it `cross_clock`, to signify that it needs the ability to accelerate at will the streaming of `music`. So, `music` is attached to `cross_clock`, and all sources built above come along. Finally, we build the fallback, which requires that all of its sources belong to the same clock. In other words, `crossfade(music)` must belong to `cross_clock` just like `safe`. The error message simply says that this is forbidden: the internal clock of our crossfade cannot be its external clock – otherwise it would not have exclusive control over its internal flow of time.

The same error also occurs on `add([crossfade(s),s])`, the simplest example of conflicting time flows, described above. However, you won't find yourself writing this obviously problematic piece of code. On the other hand, one would sometimes like to write things like our first example.

The key to the error with our first example is that the same `jingles` source is used in combination with `music` and `safe`. As a result, liquidsoap sees a potentially nasty situation, which indeed could be turned into a real mess by adding just a little more complexity. To obtain the desired effect without requiring illegal clock assignments, it suffices to create two jingle sources, one for each clock:

```
music = rotate([1,10],[playlist("jingles.lst"),
                       playlist("remote.lst")])
safe  = rotate([1,10],[playlist("jingles.lst"),
                       single("local.ogg")])
q = fallback([crossfade(music),safe])
```

There is no problem anymore: `music` belongs to `crossfade`'s internal clock, and `crossfade(music)`, `safe` and the `fallback` belong to another clock.

7.3 The clock API

There are only a couple of operations dealing explicitly with clocks.

The function `clock.assign_new(l)` creates a new clock and assigns it to all sources from the list `l`. For convenience, we also provide a wrapper, `clock(s)` which does the same with a single source instead of a list, and returns that source. With both functions, the new clock will follow (the computer's idea of) real time, unless `sync=false` is passed, in which case it will run as fast as possible.

The old (pre-1.0.0) setting `root.sync` is superseded by `clock.assign_new()`. If you want to run an output as fast as your CPU allows, just attach it to a new clock without synchronization:

```
clock.assign_new(sync=false,[output.file(%vorbis,"audio.ogg",source)])
```

This will automatically attach the appropriate sources to that clock. However, you may need to do it for other operators if they are totally unrelated to the first one.

The `buffer()` operator can be used to communicate between any two clocks: it takes a source in one clock and builds a source in another. The trick is that it uses a buffer: if one clock happens to run too fast or too slow, the buffer may empty or overflow.

Finally, `get_clock_status` provides information on existing clocks and track their respective times: it returns a list containing for each clock a pair `(name,time)` indicating the clock id its current time in *clock cycles* – a cycle corresponds to the duration of a frame, which is given in ticks, displayed on startup in the logs. The helper

function `log_clocks` built around `get_clock_status` can be used to directly obtain a simple log file, suitable for graphing with gnuplot. Those functions are useful to debug latency issues.

7.4 External clocks: decoupling latencies

The first reason to explicitly assign clocks is to precisely handle the various latencies that might occur in your setup.

Most input/output operators (ALSA, AO, Jack, OSS, etc) require their own clocks. Indeed, their processing rate is constrained by external sound APIs or by the hardware itself. Sometimes, it is too much of an inconvenience, in which case one can set `clock_safe=false` to allow another clock assignment – use at your own risk, as this might create bad latency interferences.

Currently, `output.icecast` does not require to belong to any particular clock. This allows to stream according to the soundcard's internal clock, like in most other tools: in `output.icecast(%vorbis,mount="live.ogg", input.alsa())`, the ALSA clock will drive the streaming of the soundcard input via icecast.

Sometimes, the external factors tied to Icecast output cannot be disregarded: the network may lag. If you stream a soundcard input to Icecast and the network lags, there will be a glitch in the soundcard input – a long enough lag will cause a disconnection. This might be undesirable, and is certainly disappointing if you are recording a backup of your precious soundcard input using `output.file`: by default it will suffer the same latencies and glitches, while in theory it could be perfect. To fix this you can explicitly separate Icecast (high latency, low quality acceptable) from the backup and soundcard input (low latency, high quality wanted):

```
input = input.oss()

clock.assign_new(id="icecast",
  [output.icecast(%mp3,mount="blah",mksafe(buffer(input)))])

output.file(
  %mp3,"record-%Y-%m-%d-%H-%M-%S.mp3",
  input)
```

Here, the soundcard input and file output end up in the OSS clock. The icecast output goes to the explicitly created `"icecast"` clock, and a buffer is used to connect it to the soundcard input. Small network lags will be absorbed by the buffer. Important lags and possible disconnections will result in an overflow of the buffer. In any case, the OSS input and file output won't be affected by those latencies, and the recording should be perfect. The Icecast quality is also better with that setup, since small lags are absorbed by the buffer and do not create a glitch in the OSS capture, so that Icecast listeners won't notice the lag at all.

7.5 Internal clocks: exploiting multiple cores

Clocks can also be useful even when external factors are not an issue. Indeed, several clocks run in several threads, which creates an opportunity to exploit multiple CPU cores. The story is a bit complex because OCaml has some limitations on exploiting multiple cores, but in many situations most of the computing is done in C code (typically decoding and encoding) so it parallelizes quite well.

Typically, if you run several outputs that do not share much (any) code, you can put each of them in a separate clock. For example the following script takes one file and encodes it as MP3 twice. You should run it as `liquidsoap EXPR -- FILE` and observe that it fully exploits two cores:

```
def one()
  clock.assign_new(sync=false,
        [output.file(%mp3,"/dev/null",single(argv(1)))])
```

(continues on next page)

(continued from previous page)

```
end
one()
one()
```

CHAPTER 8

A complete case analysis

We will develop here a more complex example, according to the following specifications:

- play different playlists during the day;
- play user requests – done via the telnet server;
- insert about 1 jingle every 5 songs;
- add one special jingle at the beginning of every hour, mixed on top of the normal stream;
- relay live shows as soon as one is available;
- and set up several outputs.

Once you've managed to describe what you want in such a modular way, you're half the way. More precisely, you should think of a diagram such as the following, through which the audio streams flow, following the arrows. The nodes can modify the stream using some basic operators: switching and mixing in our case. The final nodes, the ends of the paths, are outputs: they are in charge of pulling the data out of the graph and send it to the world. In our case, we only have outputs to icecast, using two different formats.

Graph for 'radio.liq'

Now here is how to write that in Liquidsoap.

```
#!/usr/bin/liquidsoap

# Lines starting with # are comments, they are ignored.

# Put the log file in some directory where
# you have permission to write.
set("log.file.path","/tmp/<script>.log")
# Print log messages to the console,
# can also be done by passing the -v option to liquidsoap.
set("log.stdout", true)
# Use the telnet server for requests
set("server.telnet", true)
```

(continues on next page)

(continued from previous page)

```
# A bunch of files and playlists,
# supposedly all located in the same base dir.

default = single("~/radio/default.ogg")

day     = playlist("~/radio/day.pls")
night   = playlist("~/radio/night.pls")
jingles = playlist("~/radio/jingles.pls")

clock   = single("~/radio/clock.ogg")
start   = single("~/radio/live_start.ogg")
stop    = single("~/radio/live_stop.ogg")

# Play user requests if there are any,
# otherwise one of our playlists,
# and the default file if anything goes wrong.
radio = fallback([ request.queue(id="request"),
                        switch([({ 6h-22h }, day),
                                ({ 22h-6h }, night)]),
                        default])
# Add the normal jingles
radio = random(weights=[1,5],[ jingles, radio ])
# And the clock jingle
radio = add([radio, switch([({0m0s},clock)])])

# Add the ability to relay live shows
full =
  fallback(track_sensitive=false,
           [input.http("http://localhost:8000/live.ogg"),
            radio])

# Output the full stream in OGG and MP3
output.icecast(%mp3,
  host="localhost",port=8000,password="hackme",
  mount="radio",full)
output.icecast(%vorbis,
  host="localhost",port=8000,password="hackme",
  mount="radio.ogg",full)

# Output the stream without live in OGG
output.icecast(%vorbis,
  host="localhost",port=8000,password="hackme",
  mount="radio_nolive.ogg",radio)
```

To try this example you need to edit the file names. In order to witness the switch from one playlist to another you can change the time intervals. If it is 16:42, try the intervals `0h-16h45` and `16h45-24h` instead of `6h-22h` and `22h-6h`. To witness the clock jingle, you can ask for it to be played every minute by using the `0s` interval instead of `0m0s`.

To try the transition to a live show you need to start a new stream on the `live.ogg` mount of your server. You can send a playlist to it using examples from the quickstart. To start a real live show from soundcard input you can use `darkice`, or simply liquidsoap if you have a working ALSA input, with:

```
liquidsoap 'output.icecast(%vorbis,
  mount="live.ogg",host="...",password="...",input.alsa())'
```

CHAPTER 9

Cookbook

The recipes show how to build a source with a particular feature. You can try short snippets by wrapping the code in an `out(..)` operator and passing it directly to liquidsoap:

```
liquidsoap -v 'out(recipe)'
```

For longer recipes, you might want to create a short script:

```
#!/usr/bin/liquidsoap -v

set("log.file.path","/tmp/<script>.log")
set("log.stdout",true)

recipe = # <fill this>
out(recipe)
```

See the quickstart guide for more information on how to run Liquidsoap, on what is this `out(..)` operator, etc.

9.1 Files

A source which infinitely repeats the same URI:

```
single("/my/default.ogg")
```

A source which plays a playlist of requests – a playlist is a file with an URI per line.

```
# Shuffle, play every URI, start over.
playlist("/my/playlist.txt")
# Do not randomize
playlist(mode="normal", "/my/pl.m3u")
# The playlist can come from any URI,
# can be reloaded every 10 minutes.
playlist(reload=600,"http://my/playlist.txt")
```

When building your stream, you'll often need to make it unfallible. Usually, you achieve that using a fallback switch (see below) with a branch made of a safe `single` or `playlist.safe`. Roughly, a single is safe when it is given a valid local audio file. A `playlist.safe` behaves just like a playlist but will check that all files in the playlist are valid local audio files. This is quite an heavy check, you don't want to have large safe playlists.

9.2 Transcoding

Liquidsoap can achieve basic streaming tasks like transcoding with ease. You input any number of "source" streams using `input.http`, and then transcode them to any number of formats / bitrates / etc. The only limitation is your hardware: encoding and decoding are both heavy on CPU. Also keep in mind a limitation inherent to OCaml: one Liquidsoap instance can only use a single processor or core. You can easily work around this limitation by launching multiple Liquidsoap instances, and thus take advantage of that 8-core Xeon server laying around in the dust in your garage.

```
# Input the stream,
# from an Icecast server or any other source
url = "http://streaming.example.com:8000/your-stream.ogg"
input = mksafe(input.http(url))

# First transcoder: MP3 32 kbps
# We also degrade the samplerate, and encode in mono
# Accordingly, a mono conversion is performed on the input stream
output.icecast(
  %mp3(bitrate=32, samplerate=22050, stereo=false),
  mount="/your-stream-32.mp3",
  host="streaming.example.com", port=8000, password="xxx",
  mean(input))
# Second transcoder : MP3 128 kbps
output.icecast(
  %mp3(bitrate=128),
  mount="/your-stream-128.mp3",
  host="streaming.example.com", port=8000, password="xxx",
  input)
```

9.3 Re-encoding a file

As a simple example using a fallible output, we shall consider re-encoding a file. We start by building a source that plays our file only once. That source is obviously fallible. We pass it to a file output, which has to be in fallible mode. We also disable the `sync` parameter on the source's clock, to encode the file as quickly as possible. Finally, we use the `on_stop` handler to shutdown liquidsoap when streaming is finished.

```
# The input file,
# any format supported by liquidsoap
input = "/tmp/input.mp3"

# The output file
output = "/tmp/output.ogg"

# A source that plays the file once
source = once(single(input))

# We use a clock with disabled synchronization
```

(continues on next page)

(continued from previous page)

```
clock.assign_new(sync=false,[source])

# Finally, we output the source to an
# ogg/vorbis file
output.file(%vorbis, output,fallible=true,
                     on_stop=shutdown,source)
```

9.4 Scheduling

```
# A fallback switch
fallback([playlist("http://my/playlist"),
          single("/my/jingle.ogg")])
# A scheduler,
# assuming you have defined the night and day sources
switch([ ({0h-7h}, night), ({7h-24h}, day) ])
```

9.5 Force a file/playlist to be played at least every XX minutes

It can be useful to have a special playlist that is played at least every 20 minutes for instance (3 times per hour). You may think of a promotional playlist for instance. Here is the recipe:

```
# (1200 sec = 20 min)
timed_promotions = delay(1200.,promotions)
main_source = fallback([timed_promotions,other_source])
```

Where promotions is a source selecting the file to be promoted.

9.6 Handle special events: mix or switch

```
# Add a jingle to your normal source
# at the beginning of every hour:
add([normal,switch([({0m0s},jingle)])])
```

Switch to a live show as soon as one is available. Make the show unavailable when it is silent, and skip tracks from the normal source if they contain too much silence.

```
stripped_stream =
  strip_blank(input.http("http://myicecast:8080/live.ogg"))
fallback(track_sensitive=false,
         [stripped_stream,skip_blank(normal)])
```

Without the `track_sensitive=false` the fallback would wait the end of a track to switch to the live. When using the blank detection operators, make sure to fine-tune their `threshold` and `length` (float) parameters.

9.7 Unix interface, dynamic requests

Liquidsoap can create a source that uses files provided by the result of the execution of any arbitrary function of your own. This is explained in the documentation for request-based sources.

For instance, the following snippet defines a source which repeatedly plays the first valid URI in the playlist:

```
request.dynamic(
  { request.create("bar:foo",
      indicators=
        get_process_lines("cat "^quote("playlist.pls"))) })
```

Of course a more interesting behaviour is obtained with a more interesting program than `cat`.

Another way of using an external program is to define a new protocol which uses it to resolve URIs. `add_protocol` takes a protocol name, a function to be used for resolving URIs using that protocol. The function will be given the URI parameter part and the time left for resolving – though nothing really bad happens if you don't respect it. It usually passes the parameter to an external program, that's how we use bubble for example:

```
add_protocol("bubble",
  fun (arg,delay) ->
    get_process_lines("/usr/bin/bubble-query "^quote(arg)))
```

When resolving the URI `bubble:artist="seed"`, liquidsoap will call the function, which will call `bubble-query 'artist="seed"'` which will output 10 lines, one URI per line.

9.8 Dynamic input with harbor

The operator `input.harbor` allows you to receive a source stream directly inside a running liquidsoap.

It starts a listening server on where any Icecast2-compatible source client can connect. When a source is connected, its input if fed to the corresponding source in the script, which becomes available.

This can be very useful to relay a live stream without polling the Icecast server for it.

An example can be:

```
# Serveur settings
set("harbor.bind_addr","0.0.0.0")

# An emergency file
emergency = single("/path/to/emergency/single.ogg")

# A playlist
playlist = playlist("/path/to/playlist")

# A live source
live = input.harbor("live",port=8080,password="hackme")

# fallback
radio = fallback(track_sensitive=false,
                 [live,playlist,emergency])

# output it
output.icecast(%vorbis, radio,mount="test",host="host")
```

This script, when launched, will start a local server, here bound to "0.0.0.0". This means that it will listen on any IP address available on the machine for a connection coming from any IP address. The server will wait for any source stream on mount point "/live" to login. Then if you start a source client and tell it to stream to your server, on port 8080, with password "hackme", the live source will become available and the radio will stream it immediately.

9.9 Adding new commands

You can add more commands to interact with your script through telnet or the server socket.

For instance, the following code, available in the standard API, attaches a `source.skip` command to a source. It is useful when the original source do not have a built-in skip command.

```
# Add a skip function to a source
# when it does not have one
# by default
def add_skip_command(s) =
 # A command to skip
 def skip(_) =
   source.skip(s)
   "Done!"
 end
 # Register the command:
 server.register(namespace="#{source.id(s)}",
                 usage="skip",
                 description="Skip the current song.",
                 "skip",skip)
end

# Attach a skip command to the source s:
add_skip_command(s)
```

9.10 Dump a stream into segmented files

It is sometimes useful (or even legally necessary) to keep a backup of an audio stream. Storing all the stream in one file can be very impractical. In order to save a file per hour in wav format, the following script can be used:

```
# A source to dump
# s = ...

# Dump the stream
file_name = '/archive/$(if $(title),"$(title)","Unknown archive")-%Y-%m-%d/%Y-%m-%d-
↪%H_%M_%S.mp3'
output.file(%mp3,filename,s)
```

This will save your source into a `mp3` file with name specified by `file_name`. In this example, we use string interpolation and time litterals to generate a different file name each time new metadata are coming from `s`.

9.11 Manually dump a stream

You may want to dump the content of a stream. The following code adds two server/telnet commands, `dump.start <filename>` and `dump.stop` to dump the content of source s into the file given as argument

```
# A source to dump
# s = (...)

# A function to stop
# the current dump source
stop_f = ref (fun () -> ())
# You should make sure you never
# do a start when another dump
# is running.

# Start to dump
def start_dump(file_name) =
  # We create a new file output
  # source
  s = output.file(%vorbis,
            fallible=true,
            on_start={log("Starting dump with file #{file_name}.ogg")},
            reopen_on_metadata=false,
            "#{file_name}",
            s)
  # We update the stop function
  stop_f := fun () -> source.shutdown(s)
end

# Stop dump
def stop_dump() =
  f = !stop_f
  f ()
end

# Some telnet/server command
server.register(namespace="dump",
                description="Start dumping.",
                usage="dump.start <filename>",
                "start",
                fun (s) ->  begin start_dump(s) "Done!" end)
server.register(namespace="dump",
                description="Stop dumping.",
                usage="dump.stop",
                "stop",
                fun (s) -> begin stop_dump() "Done!" end)
```

9.12 Transitions

There are two kinds of transitions. Transitions between two different children of a switch are not problematic. Transitions between different tracks of the same source are more tricky, since they involve a fast forward computation of the end of a track before feeding it to the transition function: such a thing is only possible when only one operator is using the source, otherwise it'll get out of sync.

9.12.1 Switch-based transitions

The switch-based operators (`switch`, `fallback` and `random`) support transitions. For every child, you can specify a transition function computing the output stream when moving from one child to another. This function is given two

`source` parameters: the child which is about to be left, and the new selected child. The default transition is `fun (a,b) -> b`, it simply relays the new selected child source.

Transitions have limited duration, defined by the `transition_length` parameter. Transition duration can be overriden by passing a metadata. Default field for it is `"liq_transition_length"` but it can also be set to a different value via the `override` parameter.

Here are some possible transition functions:

```
# A simple (long) cross-fade
# Use metadata override to make sure transition is long enough.
def crossfade(a,b)
  def add_transition_length(_) =
    [("liq_transition_length","15.")]
  end

  transition =
    add(normalize=false,
          [ sequence([ blank(duration=5.),
                       fade.in(duration=10.,b) ]),
            fade.out(duration=10.,a) ])

  # Transition can have multiple tracks so only pass the metadata
  # once.
  map_first_track(map_metadata(add_transition_length),transition)
end

# Partially apply next to give it a jingle source.
# It will fade out the old source, then play the jingle.
# At the same time it fades in the new source.
# Use metadata override to make sure transition is long enough.
def next(j,a,b)
  # This assumes that the jingle is 6 seconds long
  def add_transition_length(_) =
    [("liq_transition_length","15.")]
  end

  transition =
    add(normalize=false,
          [ sequence(merge=true,
                     [ blank(duration=3.),
                       fade.in(duration=6.,b) ]),
            sequence([fade.out(duration=9.,a),
                      j,blank()]) ])

  map_first_track(map_metadata(add_transition_length),transition)
end

# A transition, which does a cross-fading from A to B
# No need to override duration as default value (5 seconds)
# is over crossade duration (3 seconds)
def transition(j,a,b)
  add(normalize=false,
          [ fade.in(duration=3.,b),
            fade.out(duration=3.,a) ])
end
```

Finally, we build a source which plays a playlist, and switches to the live show as soon as it starts, using the `transition` function as a transition. At the end of the live, the playlist comes back with a cross-fading.

```
fallback(track_sensitive=false,
             transitions=[ crossfade, transition(jingle) ],
             [ input.http("http://localhost:8000/live.ogg"),
               playlist("playlist.pls") ])
```

9.12.2 Cross-based transitions

The `cross()` operator allows arbitrary transitions between tracks of a same source. Here is how to use it in order to get a cross-fade:

```
def crossfade(~start_next,~fade_in,~fade_out,s)
  fade.in = fade.in(duration=fade_in)
  fade.out = fade.out(duration=fade_out)
  fader = fun (_,_,_,_,a,b) -> add(normalize=false,[fade.in(b),fade.out(a)])
  cross(duration=start_next,fader,s)
end
my_source =
  crossfade(start_next=1.,fade_out=1.,fade_in=1.,my_source)
```

The `crossfade()` function is already in liquidsoap. Unless you need a custom one, you should never have to copy the above example. It is implemented in the scripting language, much like this example. You can find its code in `utils.liq`.

The fade-in and fade-out parameters indicate the duraction of the fading effects. The start-next parameters tells how much overlap there will be between the two tracks. If you want a long cross-fading with a smaller overlap, you should use a sequence to stick some blank section before the beginning of `b` in `fader`. The three parameters given here are only default values, and will be overriden by values coming from the metadata tags `liq_fade_in`, `liq_fade_out` and `liq_start_next`.

For an advanced crossfading function, you can see the crossfade documentation

9.13 Alsa unbuffered output

You can use Liquidsoap to capture and play through alsa with a minimal delay. This particulary useful when you want to run a live show from your computer. You can then directly capture and play audio through external speakers without delay for the DJ !

This configuration is not trivial since it relies on your hardware. Some hardware will allow both recording and playing at the same time, some only one at once, and some none at all.. Those note to configure are what works for us, we don't know if they'll fit all hardware.

First launch liquidsoap as a one line program

```
liquidsoap -v --debug 'input.alsa(bufferize=false)'
```

Unless you're lucky, the logs are full of lines like the following:

```
Could not set buffer size to 'frame.size' (1920 samples), got 2048.
```

The solution is then to fix the captured frame size to this value, which seems specific to your hardware. Let's try this script:

```
# Set correct frame size:
set("frame.audio.size",2048)

input = input.alsa(bufferize=false)
output.alsa(bufferize=false,input)
```

If everything goes right, you may hear on your output the captured sound without any delay ! If you want to test the difference, just run the same script with `bufferize=true` (or without this parameter since it is the default). The setting will be acknowledged in the log as follows:

```
Targetting 'frame.audio.size': 2048 audio samples = 2048 ticks.
```

If you experience problems it might be a good idea to double the value of the frame size. This increases stability, but also latency.

CHAPTER 10

Smart crossfade

10.1 Basic operator

Liquidsoap includes an advanced crossfading operator. Using it, you can code which transition you want for your songs, according to the average volume level (in dB) computed on the end of the ending track and the beginning of the new one.

The low level operator is `cross`. With it, you can register a function that returns the transition you like. The arguments passed to this function are:

- volume level for previous track
- volume level for next track
- metadata chunk for previous track
- metadata chunk for next track
- source corresponding to previous track
- source corresponding to next track

You can find its documentation in the language reference.

10.2 Example

Liquidsoap also includes a ready-to-use operator defined using `cross`, it is called `crossfade` and is defined in the pervasive helper script `utils.liq`. Its code is:

```
# Crossfade between tracks,
# taking the respective volume levels
# into account in the choice of the
# transition.
# @category Source / Track Processing
```

(continues on next page)

(continued from previous page)

```
# @param ~start_next   Crossing duration, if any.
# @param ~fade_in      Fade-in duration, if any.
# @param ~fade_out     Fade-out duration, if any.
# @param ~width        Width of the volume analysis window.
# @param ~conservative Always prepare for
#                      a premature end-of-track.
# @param s             The input source.
def crossfade (~start_next=5.,~fade_in=3.,
               ~fade_out=3., ~width=2.,
               ~conservative=false,s)
  high   = -20.
  medium = -32.
  margin = 4.
  fade.out = fade.out(type="sin",duration=fade_out)
  fade.in  = fade.in(type="sin",duration=fade_in)
  add = fun (a,b) -> add(normalize=false,[b,a])
  log = log(label="crossfade")

  def transition(a,b,ma,mb,sa,sb)

    list.iter(fun(x)->
       log(level=4,"Before: #{x}"),ma)
    list.iter(fun(x)->
       log(level=4,"After : #{x}"),mb)

    if
      # If A and B and not too loud and close,
      # fully cross-fade them.
      a <= medium and
      b <= medium and
      abs(a - b) <= margin
    then
      log("Transition: crossed, fade-in, fade-out.")
      add(fade.out(sa),fade.in(sb))

    elsif
      # If B is significantly louder than A,
      # only fade-out A.
      # We don't want to fade almost silent things,
      # ask for >medium.
      b >= a + margin and a >= medium and b <= high
    then
      log("Transition: crossed, fade-out.")
      add(fade.out(sa),sb)

    elsif
      # Do not fade if it's already very low.
      b >= a + margin and a <= medium and b <= high
    then
      log("Transition: crossed, no fade-out.")
      add(sa,sb)

    elsif
      # Opposite as the previous one.
      a >= b + margin and b >= medium and a <= high
    then
      log("Transition: crossed, fade-in.")
```

(continues on next page)

(continued from previous page)

```
      add(sa,fade.in(sb))

    # What to do with a loud end and
    # a quiet beginning ?
    # A good idea is to use a jingle to separate
    # the two tracks, but that's another story.

    else
      # Otherwise, A and B are just too loud
      # to overlap nicely, or the difference
      # between them is too large and
      # overlapping would completely mask one
      # of them.
      log("No transition: just sequencing.")
      sequence([sa, sb])
    end
  end

  cross(width=width, duration=start_next,
        conservative=conservative,
        transition,s)
end
```

You can use it directly in your script, or use this code to define yours!

CHAPTER 11

Basics

Starting with version `1.0.1`, it is possible to build a liquidsoap binary that can load all its dependencies from any arbitrary path. This is very useful to distribute a liquidsoap bundled binary, independent of the distribution used.

You can enable custom path at configure time, by passing the `--enable-custom-path` configuration option. A custom loading path is a directory that contains the following file/directories:

- `./camomile`: Camomile shared data. They are usually located in `/usr/(local/)share/camomile`
- `./libs`: pervasive scripts. Their are located in `liquidsoap/scripts` in liquidsoap's sources
- `./log`: default log directories
- `./magic`: directory for magic files. See below for more details.
- `./plugins`: default plugins directory (most likely empty)
- `./run`: default runtime files directory

CHAPTER 12

Adding liquidsoap binary

In order to ship a liquidsoap binary which is independent of the distribution it will be run on, one need to also include its dynamic libraries, except for the most common. The following command may be used to list them:

```
ldd ./liquidsoap | grep usr | cut -d' ' -f 3
```

Those libraries are usually copied into a `./ld` directory. Then, the `LD_LIBRARY_PATH` is used to point the dynamic loader to this directory.

Finally, the `liquidsoap` library is usually added in `./bin/liquidsoap`

CHAPTER 13

Configuration variables

In the following, configuration variables may refer to either absolute or relative paths. If referring to a relative path, the path is resolved relatively to the directory where the `liquidsoap` binary is located at.

In order to tell liquidsoap where its custom path is located, you need to set the `LIQUIDSOAP_BASE_DIR`.

Another important variable is `MAGIC`. It tells liquidsoap where to load the libmagic's definitions and defaults to `../magic/magic.mgc`. Older versions of libmagic may require to use `magic/magic.mime` instead.

CHAPTER 14

Full example

For a fully-functional example, you can check our heroku buildpack. Its layout is:

```
./bin
./bin/liquidsoap
./camomile
./camomile/charmaps
(...)
./ld
./ld/libao.so.2
(...)
./libs
./libs/externals.liq
(...)
./log
./magic
./magic/magic.mime
./plugins
./run
```

Its configuration variables are set to:

```
LD_LIBRARY_PATH=/path/to/ld
LIQUIDSOAP_BASE_DIR=..
MAGIC=../magic/magic.mime
```

As you can see, we use an old version of `libmagic` so we need to load `magic.mime` instead of `magic.mgc`.

CHAPTER 15

Encoding formats

Encoders are used to define formats into which raw sources should be encoded by an output. Syntax for encoder is: `%encoder(parameters...)` or, if you use default parameters, `%encoder`.

15.1 Formats determine the stream content

In most liquidsoap scripts, the encoding format determines what kind of data is streamed.

The type of an encoding format depends on its parameter. For example, `%mp3` has type `format(audio=2,video=0,midi=0)` but `%mp3(mono)` has type `format(audio=1,video=0,midi=0)`.

The type of an output like `output.icecast` or `output.file` is something like `(...,format('a),...,source('a))->source('a)`. This means that your source will have to have the same type as your format.

For example if you write

```
output.file(%mp3,"/tmp/foo.mp3",playlist("~/audio"))
```

then the playlist source will have to stream stereo audio. Thus it will reject mono and video files.

Liquidsoap provides operators that can be used to convert sources into a format acceptable for a given encoder. For instance, the `mean` operator transforms any audio source into a mono source and the `audio_to_stereo` operator transforms any audio source into a stereo source.

15.2 Format variables (or lack of, rather..)

You can store an atomic format in a variable, it is a value like another: `fmt = %mp3`. However, an atomic format is an atomic constant despite its appearance. You cannot use a variable for one of its parameters: for example

```
x = 44100
%vorbis(samplerate=x)
```

is not allowed, you must write `%vorbis(samplerate=44100)`.

If you really need to use variables in encoder, for instance if bitrate is given by a user's configuration, you may alleviate that by generating a pre-defined list of possible encoders and include it on top of your script using the `%include` directive.

CHAPTER 16

List of formats and their syntax

All parameters are optional, and the parenthesis are not needed when no parameter is passed. In the following default values are shown. As a special case, the keywords `mono` and `stereo` can be used to indicate the number of channels (whether is is passed as an integer or a boolean).

16.1 MP3

Mp3 encoder comes in 3 flavors:

- `%mp3` or `%mp3.cbr`: Constant bitrate encoding
- `%mp3.vbr`: Variable bitrate, quality-based encoding.
- `%mp3.abr`: Average bitrate based encoding.

Parameters common to each flavor are:

- `stereo=true/false`, `mono=true/false`: Encode stereo or mono data (default: `stereo`).
- `stereo_mode`: One of: `"stereo"`, `"joint_stereo"` or `"default"` (default: `"default"`)
- `samplerate=44100`: Encoded data samplerate (default: `44100`)
- `internal_quality=2`: Lame algorithms internal quality. A value between `0` and `9`, `0` being highest quality and `9` the worst (default: `2`).
- `id3v2=true`: Add an `id3v2` tag to encoded data (default: `false`). This option is only valid if liquidsoap has been compiled with taglib support.

Parameters for `%mp3` are:

- `bitrate`: Encoded data fixed bitrate

Parameters for `%mp3.vbr` are:

- `quality`: Quality of encoded data; ranges from `0` (highest quality) to `9` (worst quality).

Parameters for `%mp3.abr` are:

- `bitrate`: Average bitrate
- `min_bitrate`: Minimun bitrate
- `max_bitrate`: Maximun bitrate
- `hard_min`: Enforce minimal bitrate

Examples:

- Contstant `128` kbps bitrate encoding: `%mp3(bitrate=128)`
- Variable bitrate with quality `6` and samplerate of `22050` Hz: `%mp3.vbr(quality=7, samplerate=22050)`
- Average bitrate with mean of `128` kbps, maximun bitrate `192` kbps and `id3v2` tags: `%mp3.abr(bitrate=128,max_bitrate=192,id3v2=true)`

Optionally, liquidsoap can insert a message within mp3 data. You can set its value using the `msg` parameter. Setting it to `""` disables this feature. This is its default value.

16.2 Shine

Shine is the fixed-point mp3 encoder. It is useful on architectures without a FPU, such as ARM. It is named `%shine` or `%mp3.fxp` and its parameters are:

```
%shine(channels=2,samplerate=44100,bitrate=128)
```

16.3 WAV

```
%wav(stereo=true, channels=2, samplesize=16, header=true, duration=10.)
```

If `header` is `false`, the encoder outputs raw PCM. `duration` is optional and is used to set the WAV length header.

Because Liquidsoap encodes a possibly infinite stream, there is no way to know in advance the duration of encoded data. Since WAV header has to be written first, by default its length is set to the maximun possible value. If you know the expected duration of the encoded data and you actually care about the WAV length header then you should use this parameter.

16.4 FFmpeg

The `%ffmpeg` encoder is the latest addition to our collection, starting with version `1.4.1`. It is only for audio encoding for now. You need to have ocaml-ffmpeg installed and up-to date to enable the encoder during liquidsoap's build.

The encoder should support all the options for `ffmpeg`'s muxers and encoders, including private configuration options. Configuration value are passed as key/values, with values being of types: `string`, `int`, or `float`. If a configuration is not recognized (or: unused), it will raise an error during the instantiation of the encoder. Here are some configuration examples:

- **AAC encoding at `22050kHz` using `fdk-aac` encoder and `mpegts` muxer**

```
%ffmpeg(format="mpegts",ar=22050,codec="libfdk_aac",b="32k",afterburner=1,profile=
↪"aac_he_v2")
```

- **Mp3 encoding using `libshine`**

```
%ffmpeg(format="mp3",codec="libshine")
```

The `%ffmpeg` encoder is the prime encoder for HLS output as it is the only one of our collection of encoder which can produce Mpeg-ts muxed data, which is required by most HLS clients.

16.5 Ogg

The following formats can be put together in an Ogg container. The syntax for doing so is `%ogg(x,y,z)` but it is also possible to just write `%vorbis(...)`, for example, instead of `%ogg(%vorbis(...))`.

All ogg encoders have a `bytes_per_page` parameter, which can be used to try to limit ogg logical pages size. For instance:

```
# Try to limit vorbis pages size to 1024 bytes
%vorbis(bytes_per_page=1024)
```

16.5.1 Vorbis

```
# Variable bitrate
%vorbis(samplerate=44100, channels=2, quality=0.3)
% Average bitrate
%vorbis.abr(samplerate=44100, channels=2, bitrate=128, max_bitrate=192, min_
↪bitrate=64)
# Constant bitrate
%vorbis.cbr(samplerate=44100, channels=2, bitrate=128)
```

Quality ranges from -0.2 to 1, but quality -0.2 is only available with the aotuv implementation of libvorbis.

16.5.2 Opus

Opus is a lossy audio compression made especially suitable for interactive real-time applications over the Internet. Liquidsoap supports Opus data encapsulated into Ogg streams.

The encoder is named `%opus` and its parameters are as follows. Please refer to the Opus documentation for information about their meanings and values.

- `vbr`: one of `"none"`, `"constrained"` or `"unconstrained"`
- `application`: One of `"audio"`, `"voip"` or `"restricted_lowdelay"`
- `complexity`: Integer value between `0` and `10`.
- `max_bandwidth`: One of `"narrow_band"`, `"medium_band"`, `"wide_band"`, `"super_wide_band"` or `"full_band"`
- `samplerate`: input samplerate. Must be one of: `8000`, `12000`, `16000`, `24000` or `48000`
- `frame_size`: encoding frame size, in milliseconds. Must be one of: `2.5`, `5.`, `10.`, `20.`, `40.` or `60.`.
- `bitrate`: encoding bitrate, in `kbps`. Must be a value between `5` and `512`. You can also set it to `"auto"`.
- `channels`: currently, only `1` or `2` channels are allowed.
- `mono`, `stereo`: equivalent to `channels=1` and `channels=2`.

- `signal`: one of `"voice"` or `"music"`

16.5.3 Theora

```
%theora(quality=40,width=640,height=480,
        picture_width=255,picture_height=255,
        picture_x=0, picture_y=0,
        aspect_numerator=1, aspect_denominator=1,
        keyframe_frequency=64, vp3_compatible=false,
        soft_target=false, buffer_delay=5,
        speed=0)
```

You can also pass `bitrate=x` explicitly instead of a quality. The default dimensions are liquidsoap's default, from the settings `frame.video.height/width`.

16.5.4 Speex

```
%speex(stereo=false, samplerate=44100, quality=7,
       mode=wideband, # One of: wideband|narrowband|ultra-wideband
       frames_per_packet=1,
       complexity=5)
```

You can also control quality using `abr=x` or `vbr=y`.

16.5.5 Flac

The flac encoding format comes in two flavors:

- `%flac` is the native flac format, useful for file output but not for streaming purpose
- `%ogg(%flac,...)` is the ogg/flac format, which can be used to broadcast data with icecast

The parameters are:

```
%flac(samplerate=44100,
      channels=2,
      compression=5,
      bits_per_sample=16)
```

`compression` ranges from 0 to 8 and `bits_per_sample` should be one of: `8`, `16` or `32`.

16.6 FDK-AAC

This encoder can do both AAC and AAC+.

Its syntax is:

```
%fdkaac(channels=2, samplerate=44100, bandwidth="auto", bitrate=64, afterburner=false,
↪ aot="mpeg2_he_aac_v2", transmux="adts", sbr_mode=false)
```

Where `aot` is one of: `"mpeg4_aac_lc"`, `"mpeg4_he_aac"`, `"mpeg4_he_aac_v2"`, `"mpeg4_aac_ld"`, `"mpeg4_aac_eld"`, `"mpeg2_aac_lc"`, `"mpeg2_he_aac"` or `"mpeg2_he_aac_v2"`

`bandwidth` is one of: `"auto"`, any supported integer value.

`transmux` is one of: `"raw"`, `"adif"`, `"adts"`, `"latm"`, `"latm_out_of_band"` or `"loas"`.

Bitrate can be either constant by passing: `bitrate=64` or variable: `vbr=<1-5>`

You can consult the Hydrogenaudio knowledge base for more details on configuration values and meanings.

16.7 Gstreamer

The `%gstreamer` encoder can be used to encode streams using the `gstreamer` multimedia framework. This encoder extends liquidsoap with all available GStreamer formats which includes most, if not all, formats available to your operating system.

The encoder's parameters are as follows:

```
%gstreamer(channels=2,
           audio="lamemp3enc",
           has_video=true,
           video="x264enc",
           muxer="mpegtsmux",
           metadata="metadata",
           log=5,
           pipeline="")
```

Please refer to the Gstreamer encoder page for a detailed explanation of this encoder.

16.8 External encoders

For a detailed presentation of external encoders, see this page.

```
%external(channels=2,samplerate=44100,header=true,
          restart_on_crash=false,
          restart_on_metadata,
          restart_after_delay=30,
          process="progname")
```

Only one of `restart_on_metadata` and `restart_after_delay` should be passed. The delay is specified in seconds. The encoding process is mandatory, and can also be passed directly as a string, without `process=`.

CHAPTER 17

Introduction

You can use external programs in liquidsoap to decode audio files. The program must be able to output WAV data to its standard output (`stdout`) and, posssibly, read encoded data from its standard input.

Please note that this feature is not available under Windows.

CHAPTER 18

Basic operators

External decoders are registered using the `add_decoder` and `add_oblivious_decoder` operators. They are invoked the following way:

18.1 add_decoder

```
add_decoder(name="my_decoder",description="My custom decoder",
            test,decoder)
```

`add_decoder` is used for external decoders that can read the encoded data from their standard input (stdin) and write the decoded data as WAV to their standard output (stdout). This operator is recommended because its estimation of the remaining time is better than the estimation done by the decoders registered using `add_oblivious_decoder`. The important parameters are:

- `test` is a function used to determine if the file should be decoded by the decoder. Returned values are: * `0`: no decodable audio,
- `-1`: decodable audio but number of audio channels unknown,
- `x`: fixed number of decodable audio channels.
- `decoder` is the string containing the shell command to run to execute the decoding process.

18.2 add_oblivious_decoder

`add_oblivious_decoder` is very similar to `add_decoder`. The main difference is that the decoding program reads encoded data directly from the local files and not its standard input. Decoders registered using this operator do not have a reliable estimation of the remaining time. You should use `add_oblivious_decoder` only if your decoding program is not able to read the encoded data from its standard input.

```
add_oblivious_decoder(name="my_decoder",description="My custom decoder",
                      buffer=5., test,decoder)
```

`add_decoder` is used for external decoders that can read the encoded data from their standard input (stdin) and write the decoded data as WAV to their standard output (stdout). This operator is recommended because its estimation of the remaining time is better than the estimation done by the decoders registered using `add_oblivious_decoder`. The important parameters are:

- `test` is a function used to determine if the file should be decoded by the decoder. Returned values are: * `0`: no decodable audio,
- `-1`: decodable audio but number of audio channels unknown,
- `x`: fixed number of decodable audio channels.
- `decoder` is a function that receives the name of the file that should be decoded and returns a string containing the shell command to run to execute the decoding process.

18.3 add_metadata_resolver

You may also register new metadata resolvers using the `add_metadata_resolver` operator. It is invoked the following way: `add_metadata_resolver(format,resolver)`, where:

- `format` is the name of the resolved format. It is only informative.
- `resolver` is a function `f` that returns a list of metadata of the form: `(label, value)`. It is invoked the following way: `f(format=name,file)`, where: * `format` contains the name of the format, as returned by the decoder that accepted to decode the file. `f` may return immediately if this is not an expected value.
- `file` is the name of the file to decode.

CHAPTER 19

Wrappers

On top of the basic operators, wrappers have been written for some common decoders. This includes the `flac` and `faad` decoders, by default. All the operators are defined in `externals.liq`.

19.1 The FLAC decoder

The flac decoder uses the `flac` command line. It is enabled if the binary can be found in the current `$PATH`.

Its code is the following:

```
def test_flac(file) =
  if test_process("which metaflac") then
    channels = list.hd(default="",get_process_lines("metaflac \
                                          --show-channels #{quote(file)} \
                                          2>/dev/null"))
    # If the value is not an int, this returns 0 and we are ok :)
    int_of_string(channels)
  else
    # Try to detect using mime test..
    mime = get_mime(file)
    if string.match(pattern="flac",file) then
      # We do not know the number of audio channels
      # so setting to -1
      (-1)
    else
      # All tests failed: no audio decodable using flac..
      0
    end
  end
end
add_decoder(name="FLAC",description="Decode files using the flac \
            decoder binary.", test=test_flac,flac_p)
```

Additionaly, a metadata resolver is registered when the `metaflac` command can be found in the `$PATH`:

```
if test_process("which metaflac") then
  log(level=3,"Found metaflac binary: \
               enabling flac external metadata resolver.")
  def flac_meta(file)
    ret = get_process_lines("metaflac --export-tags-to=- \
                             #{quote(file)} 2>/dev/null")
    ret = list.map(string.split(separator="="),ret)
    # Could be made better..
    def f(l',l)=
      if list.length(l) >= 2 then
        list.append([(list.hd(default="",l),list.nth(default="",l,1))],l')
      else
        if list.length(l) >= 1 then
          list.append([(list.hd(default="",l),"")],l')
        else
          l'
        end
      end
    end
  list.fold(f,[],ret)
  end
  add_metadata_resolver("FLAC",flac_meta)
end
```

19.2 The faad decoder

The faad decoder uses the `faad` program, if found in the `$PATH`. It can decode AAC and AAC+ audio files. This program does not support reading encoded data from its standard input so the decoder is registered using `add_oblivious_decoder`.

Its code is the following:

```
  aac_mimes = ["audio/aac", "audio/aacp", "audio/3gpp", "audio/3gpp2", "audio/mp4",
               "audio/MP4A-LATM", "audio/mpeg4-generic", "audio/x-hx-aac-adts"]
  aac_filexts = ["m4a", "m4b", "m4p", "m4v",
                 "m4r", "3gp", "mp4", "aac"]

  # Faad is not very selective so
  # We are checking only file that
  # end with a known extension or mime type
  def faad_test(file) =
    # Get the file's mime
    mime = get_mime(file)
    # Test mime
    if list.mem(mime,aac_mimes) then
      true
    else
      # Otherwise test file extension
      ret = string.extract(pattern='\.(.+)$',file)
        if list.length(ret) != 0 then
          ext = ret["1"]
          list.mem(ext,aac_filexts)
        else
          false
        end
```

(continues on next page)

(continued from previous page)

```
    end
  end

  if test_process("which faad") then
    log(level=3,"Found faad binary: enabling external faad decoder and \
                 metadata resolver.")
    faad_p = (fun (f) -> "faad -w #{quote(f)} 2>/dev/null")
    def test_faad(file) =
      if faad_test(file) then
        channels = list.hd(default="",get_process_lines("faad -i #{quote(file)} 2>&1␣
↪| \
                                                         grep 'ch,'"))
        ret = string.extract(pattern=", (\d) ch,",channels)
        ret =
          if list.length(ret) == 0 then
          # If we pass the faad_test, chances are
          # high that the file will contain aac audio data..
            "-1"
          else
            ret["1"]
          end
        int_of_string(default=(-1),ret)
      else
        0
      end
    end
    add_oblivious_decoder(name="FAAD",description="Decode files using \
                          the faad binary.", test=test_faad, faad_p)
    def faad_meta(file) =
      if faad_test(file) then
        ret = get_process_lines("faad -i \
                    #{quote(file)} 2>&1")
        # Yea, this is tuff programming (again) !
        def get_meta(l,s)=
          ret = string.extract(pattern="^(\w+):\s(.+)$",s)
          if list.length(ret) > 0 then
            list.append([(ret["1"],ret["2"])],l)
          else
            l
          end
        end
        list.fold(get_meta,[],ret)
      else
        []
      end
    end
    add_metadata_resolver("FAAD",faad_meta)
  end
```

CHAPTER 20

Introduction

You can use any external program that accepts wav or raw PCM data to encode audio data and use the resulting compressed stream as an output, either to a file, a pipe, or even icecast.

When using an external encoding process, uncompressed PCM data will be sent to the process through its standard input (`stdin`), and encoded data will be read through its standard output (`stdout`). When using a process that does only file input or output, `/dev/stdin` and `/dev/stdout` can be used, though this may generate issues if the encoding process expects to be able to go backward/forward in the file.

CHAPTER 21

External encoders

The main operators that can be used with external encoders are:

- `output.file`
- `output.icecast`

In order to use external encoders with these operators, you have to use the `%external` encoding format. Its syntax is:

```
%external(channels=2,samplerate=44100,header=true,
          restart_on_crash=false,
          restart_on_metadata,
          restart_after_delay=30,
          process="progname")
```

The available options are:

- `process`: this parameter is a function that takes the current metadata and return the process to start.
- `header`: if set to `false` then no WAV header will be added to the data fed to the encoding process, thus the encoding process shall operate on RAW data.
- `restart_on_crash`: wether to restart the encoding process if it crashed. Useful when the external process fails to encode properly data after some time.
- `restart_on_metadata`: restart encoding process on each new metadata:. Useful in conjonction with the `process` parameter for audio formats that need a new header, possibly with metadatas, for each new track. This is the case for the ogg container.
- `restart_encoder_delay`: Restart the encoder after some delay. This can be useful for encoders that cannot operate on infinite streams, or are buggy after some time, like the `lame` binary. The default for `lame` and `accplusenc`-based encoders is to restart the encoder every hour.

Only one of `restart_encoder_delay` or `restart_on_new_track` should be used.

The restart mechanism strongly relies on the good behaviour of the encoding process. The restart operation will close the standard input of the encoding process. The encoding process is then expected to finish its own operations and close its standard output. If it does not close its standard output, the encoding task will not finish.

If your encoding process has this issue, you should turn the `restart_on_crash` option to `true` and kill the encoding process yourself.

If you use an external encoder with the `output.icecast` operator, you should also use the following options of `output.icecast`:

- `icy_metadata`: send new metadata as ICY update. This is the case for headerless formats, such as MP3 or AAC, and it appears to work also for ogg/vorbis streams.
- `format`: Content-type (mime) of the data sent to icecast. For instance, for ogg data, it is one of `application/ogg'',` audio/ogg" or `video/ogg'' and for mp3 data it is` audio/mpeg".

CHAPTER 22

Video support

Videos can also be encoded by programs able to read files in avi format from standard input. To use it, the flag `video=true` of `%external` should be used. For instance, a compresed avi file can be generated with `ffmpeg` using

```
output.file(
  %external(process="ffmpeg -i pipe:0 -f avi pipe:1",video=true),
  "/tmp/test.avi", s)
```

CHAPTER 23

Introduction

You can use an external program to create a source that will read data coming out of the standard output (`stdout`) of this program. Contrary to the external file decoders, data will be buffered and played when a sufficient amount was accumulated.

The program should output data in signed 16 bits little endian PCM (s16le). Number of channels and samplerate can be specified. There is no need of any wav header in the data, though it should work too.

CHAPTER 24

Basic operator

The basic operator for creating an external stream is `input.external`. Its parameters are:

- `buffer`: Duration of the pre-buffered data.
- `max`: Maximum duration of the buffered data.
- `channels`: Number of channels.
- `samplerate`: Sample rate.
- `restart`: Restart the process when it has exited normaly.
- `restart_on_error`: Restart the process when it has exited with error.

The last parameter is unlabeled. It is a string containing the command that will be executed to run the external program.

CHAPTER 25

Wrappers

A wrapper, `input.mplayer`, is defined to use mplayer as the external decoder. Its code is:

```
# Stream data from mplayer
# @category Source / Input
# @param s data URI.
# @param ~restart restart on exit.
# @param ~restart_on_error restart on exit with error.
# @param ~buffer Duration of the pre-buffered data.
# @param ~max Maximum duration of the buffered data.
def input.mplayer(~id="input.mplayer",
         ~restart=true,~restart_on_error=false,
         ~buffer=0.2,~max=10.,s) =
  input.external(id=id,restart=restart,
      restart_on_error=restart_on_error,
      buffer=buffer,max=max,
      "mplayer -really-quiet \
              -ao pcm:file=/dev/stdout \
              -vc null -vo null #{quote(s)} \
                    2>/dev/null")
end
```

CHAPTER 26

Frequently Asked Questions

26.1 What does this message means?

26.1.1 Type error

Liquidsoap might also reject a script with a series of errors of the form `this value has type ... but it should be a subtype of ...`. Usually the last error tells you what the problem is, but the previous errors might provide a better information as to where the error comes from.

For example, the error might indicate that a value of type `int` has been passed where a float was expected, in which case you should use a conversion, or more likely change an integer value such as `13` into a float `13.`.

A type error can also show that you're trying to use a source of a certain content type (e.g., audio) in a place where another content type (e.g., pure video) is required. In that case the last error in the list is not the most useful one, but you will read something like this above:

```
At ...:
  this value has type
    source(audio=?A+1,video=0,midi=0)
    where ?A is a fixed arity type
  but it should be a subtype of
    source(audio=0,video=1,midi=0)
```

Sometimes, the type error actually indicates a mistake in the order or labels of arguments. For example, given `output.icecast(mount="foo.ogg",source)` liquidsoap will complain that the second argument is a source (`source(?A)`) but should be a format (`format(?A)`): indeed, the first unlabelled argument is expected to be the encoding format, e.g., `%vorbis`, and the source comes only second.

Finally, a type error can indicate that you have forgotten to pass a mandatory parameter to some function. For example, on the code `fallback([crossfade(x),...])`, liquidsoap will complain as follows:

```
At line ...:
  this value has type
    (?id:string, ~start_next:float, ~fade_in:float,
```

(continues on next page)

(continued from previous page)

```
    ~fade_out:float)->source(audio=?A,video=?B,midi=0)
   where ?B, ?A is a fixed arity type
 but it should be a subtype of
   source(audio=?A,video=?B,midi=0)
   where ?B, ?A is a fixed arity type
```

Indeed, `fallback` expects a source, but `crossfade(x)` is still a function expecting the parameters `start_next`, `fade_in` and `fade_out`.

26.1.2 That source is fallible!

See the quickstart, or read more about sources.

26.1.3 Clock error

Read about clocks for the errors `a source cannot belong to two clocks` and `cannot unify two nested clocks`.

26.1.4 We must catchup x.xx!

This error means that a clock is getting late in liquidsoap. This can be caused by an overloaded CPU, if your script is doing too much encoding or processing: in that case, you should reduce the load on your machine or simplify your liquidsoap script. The latency may also be caused by some lag, for example a network lag will cause the icecast output to hang, making the clock late.

The first kind of latency is problematic because it tends to accumulate, eventually leading to the restarting of outputs:

```
Too much latency!
Resetting active source...
```

The second kind of latency can often be ignored: if you are streaming to an icecast server, there are several buffers between you and your listeners which make this problem invisible to them. But in more realtime applications, even small lags will result in glitches.

In some situations, it is possible to isolate some parts of a script from the latency caused by other parts. For example, it is possible to produce a clean script and back it up into a file, independently of its output to icecast (which again is sensitive to network lags). For more details on those techniques, read about clocks.

26.1.5 Unable to decode “file” as {audio=2;video=0;midi=0}!

This log message informs you that liquidsoap failed to decode a file, not necessarily because it cannot handle the file, but also possibly because the file does not contain the expected media type. For example, if video is expected, an audio file will be rejected.

The case of mono files is often surprising. Since liquidsoap does not implicitly convert between media formats, input files must be stereo if the output expects stereo data. As a result, people often get this error message on files which they expected to play correctly. The simple way to fix this is to use the `audio_to_stereo()` operator to allow any kind of audio on its input, and produce stereo as expected on its output.

26.1.6 Exceptions

Liquidsoap dies with messages such as these by the end of the log:

```
... [threads:1] Thread "XXX" aborts with exception YYY!
... [stderr:3] Thread 2 killed on uncaught exception YYY.
... [stderr:3] Raised at file ..., line ..., etc.
```

Those internal errors can be of two sorts:

- **Bug**: Normally, this means that you've found a bug, which you should report on the mailing list or bug tracker.
- **User error**: In some cases, we let an exception go on user errors, instead of nicely reporting and handling it. By looking at the surrounding log messages, you might realize that liquidsoap crashed for a good reason, that you are responsible for fixing. You can still report a bug: you should not have seen an exception and its backtrace.

In any case, once that kind of error happens, there is no way for the user to prevent liquidsoap from crashing. Those exceptions cannot be caught or handled in any way at the level of liquidsoap scripts.

26.2 Troubleshooting

26.2.1 Pulseaudio

When using ALSA input or output or, more generaly any audio input or output that is not using pulseaudio, you should disable pulseaudio, which is often installed by default. Pulseaudio emulates ALSA but this also generates bugs, in particular errors of this form:

```
Alsa.Unknown_error(1073697252)!
```

There are two things you may do:

- Make sure your alsa input/output does not use pulseaudio
- Disable pulseaudio on your system

In the first case, you should first find out which sound card you want to use, with the command `aplay -l`. An example of its output is:

```
**** List of PLAYBACK Hardware Devices ****
card 0: Intel [HDA Intel], device 0: STAC92xx Analog [STAC92xx Analog]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

In this case, the card we want to use is: device `0`, subdevice `0`, thus: `hw:0,0`. We now create a file `/etc/asound.conf` (or `~/.asoundrc` for single-user configuration) that contains the following:

```
pcm.liquidsoap {
        type plug
        slave { pcm "hw:0,0" }
}
```

This creates a new alsa device that you can use with liquidsoap. The `plug` operator in ALSA is used to work-around any hardward limitations in your device (mixing multiple outputs, resampling etc.). In some cases you may need to read more about ALSA and define your own PCM device.

Once you have created this device, you can use it in liquidsoap as follows:

```
input.alsa(device="pcm.liquidsoap", ...)
```

In the second case – disabling pulseaudio, you can edit the file `/etc/pulse/client.conf` and change or add this line:

```
autospawn = no
```

And kill any running pulseaudio process:

```
killall pulseaudio
```

Otherwise you may simply remove pulseaudio's packages, if you use Debian or Ubuntu:

```
apt-get remove pulseaudio libasound2-plugins
```

26.2.2 Listeners are disconnected at the end of every track

Several media players, including renowned ones, do not properly support Ogg/Vorbis streams: they treat the end of a track as an end of file, resulting in the disconnection.

Players that are affected by this problem include VLC. Players that are not affected include ogg123, liquidsoap.

One way to work around this problem is to not use Ogg/Vorbis (which we do not recommend) or to not produce tracks within a Vorbis stream. This is done by merging liquidsoap tracks (for example using `add(normalize=false, [blank(),source])`) and also not passing any metadata (which is also a result of the previous snippet).

26.2.3 Encoding blank

Encoding pure silence is often too effective for streaming: data is so compressed that there is nothing to send to listeners, whose clients eventually disconnect. Therefore, it is a good idea to use a non-silent jingle instead of `blank()` to fill in the blank. You can also achieve various effects using synthesis sources such as `noise()`, `sine()`, etc.

26.2.4 Temporary files

Liquidsoap relies on OCaml's `Filename.tmp_dir_name` variable to store temporary files. It is documented as follows:

The name of the temporary directory: Under Unix, the value of the `TMPDIR` environment variable, or `"/tmp"` if the variable is not set. Under Windows, the value of the `TEMP` environment variable, or `"."` if the variable is not set.

CHAPTER 27

Developping Flows

Flows is handled on the Heroku platform.

27.1 Getting started

First steps to get started.

- Create an account on Heroku.
- Install the Heroku utilities.
- Ask a Liquidsoap administrator to give you access to the repositories.

The repositories of the main components are organized as follows.

- `savonet-flows` is the python handler to submit metadata:
  - the associated github repository
  - the Heroku repository is `git@heroku.com:savonet-liquidsoap.git`
- `savonet-flows-socket` is the node application to serve the webpage and client stuff.
  - the associated github repository
  - the test Heroku webpage is updated by pushing on `git@heroku.com:savonet-flows-socket-next.git`
  - the prod Heroku webpage is updated by pushing on `git@heroku.com:savonet-flows-socket.git`

Some more experimental repositories include:

- a command-line client

27.2 Useful commands

Getting the environment variables:

```
heroku config -s --app savonet-flows
```

Seeing the logs of the socket application:

```
heroku logs -t --app savonet-flows-socket
```

CHAPTER 28

Flows

We maintain a webpage of streams generated by Liquidsoap. In order to register your radio on this page, a simple operator called `register_flow` is provided. If your stream is called `stream`, just wrap it as follows before outputting it:

```
stream = register_flow(
  radio="My radio",
  website="http://my.web.site/",
  description="The sound of my computer",
  genre="reggae",
  user="foo",
  password="bar",
  streams=[("mp3/128k","http://my.web.site/stream")],
  stream)
```

Most parameters are pretty explicit: `radio` is the name of the radio, `website` is the url of the website of the radio, etc.

The parameter `streams` is a list of pairs of strings describing the format of the stream and the url of the stream. We use a list here because one radio can be broadcasted under multiple stream formats. The format should always be of the form `codec/bitrate` (e.g. `mp3/128k`) or `codec` if it is encoded in variable bitrate.

You can register a stream with any `user` and `password` parameters. Those parameters are only useful if you use additional services such as the command-line client, or if you want to claim that you ‘‘own” a registered radio. All this is relatively informal for now, but may be tightened in future developments of Flows.

28.1 Getting information from flows

If you develop a website and registered your radio as explained above, flows can be used as a convenient way to retrieve metadata in javascript, get a list of friendly radios, etc. The flows service currently consists of a list of radios broadcasting using Liquidsoap, with their corresponding streams and metadata.

28.2 List of radios

Note: When the list gets big enough, this API should be changed to return paginated results.

A list of radios, encoded in JSON format, can be obtained by querying the following url:

```
http://flows.liquidsoap.info/radios
```

Output is a JSON string like this:

```
[{ "token"       : "a60f5cadf2645321d4d061896318a2d99f2ff6a6",
   "name"        : "RadioPi - Canal Jazz",
   "website"     : "http://www.radiopi.org/",
   "description" : "Cool vibes from Chatenay!",
   "genre"       : "jazz",
   "longitude"   : 2.26670002937317,
   "latitude"    : 48.7667007446289,
   "title"       : "Bud Powell - Un Poco Loco",
   "artist"      : "Va-1939-1999 - 60 Ans De Jazz",
   "streams"     : [{ "format" : "mp3/128k",
                      "url"    : "http://radiopi.org:8080/jazz" }]}
 ...
```

28.3 Single radio

A single radio, encoded in JSON format, can be obtained by querying the following url:

```
http://flows.liquidsoap.info/radio?name=foo&website=bar
```

All argument are optional and should be in UTF8 and properly encoded for a HTTP GET request.

A direct request using a radio's token can also be performed at this URL:

```
http://flows.liquidsoap.info/radio/:token
```

Output is a JSON string like this:

```
{ "token"       : "a60f5cadf2645321d4d061896318a2d99f2ff6a6",
  "name"        : "RadioPi - Canal Jazz",
  "website"     : "http://www.radiopi.org/",
  "description" : "Cool vibes from Chatenay!",
  "genre"       : "jazz",
  "longitude"   : 2.26670002937317,
  "latitude"    : 48.7667007446289,
  "title"       : "Bud Powell - Un Poco Loco",
  "artist"      : "Va-1939-1999 - 60 Ans De Jazz",
  "streams"     : [{ "format" : "mp3/128k",
                     "url"    : "http://radiopi.org:8080/jazz" }]}
```

28.4 Stream redirection

Flows can also be used to provide a single url for all your listeners, allowing you to change the actual listening url of your radio while preserving your playlists.

If you radio's token is `:token` and has a stream of format `:format`, then the following url will redirect any request to your stream's URL.

```
http://flows.liquidsoap.info/radio/:token/:format
```

28.5 Playlist

As for streams, if you radio's token is `:token` then the following link will return a PLS playlist:

```
http://flows.liquidsoap.info/radio/:token.pls
```

28.6 Real-time notifications

It is oftentimes difficult to properly implement a regular update of your radio's currently playing information on a website or an online player. Using Flows, you can very easily implement real-time notifications of the song currently being played by your radio.

First, you need to know your radio's token. You can do so by querying a single radio, as described above, or querying all radios and finding yours in the list.

Then, in your webpage's head, you need to add javascript code adapted from this example:

```
<script src="http://flows.liquidsoap.info/socket.io/socket.io.js"></script>
<script type="text/javascript">
  var socket = io.connect("http://flows.liquidsoap.info");

  socket.emit('join', radio_token);

  socket.on('joined', function (radio) {
    console.log("Successfully joined " + radio.name + " notification channel. Current␣
↪title is: " + radio.title + ".");
  });

  socket.on('error', function (data) {
    console.log("Error: " + data + ".");
  });

  socket.on(radio_token, function (data) {
    console.log("Got new notification " + data.cmd + " for radio " + data.radio.name␣
↪+ ": " + JSON.stringify(data));
  });
</script>
```

Received messages are objects of the form:

```
{ cmd   : "metadata",
  radio : { token       : "a60f5cadf2645321d4d061896318a2d99f2ff6a6",
            name        : "RadioPi - Canal Jazz",
            website     : "http://www.radiopi.org/",
            description : "Cool vibes from Chatenay!",
            genre       : "jazz",
            longitude   : 2.26670002937317,
            latitude    : 48.7667007446289,
```

(continues on next page)

(continued from previous page)

```
            title       : "Bud Powell - Un Poco Loco",
            artist      : "Va-1939-1999 - 60 Ans De Jazz",
            streams     : [{ format : "mp3/128k",
                             url    : "http://radiopi.org:8080/jazz" }]}}
```

Messages with command `"metadata"` are the one you want to use to update information displayed about your radio.

28.7 Twitter notifications

Note: The twitter API is deprecated for now.

You can register twitter notifications on any twitter account with radio that you own. The API is designed to allow implementing this feature on your website. It works as follows:

First you issue a `HTTP GET` request, authenticated with your radio credentials at this address:

```
http://flows.liquidsoap.info/radio/:token/twitter/auth?redirect_to=:link
```

Then you should receive a response of the form:

```
{"url":"https://api.twitter.com/oauth/authenticate?oauth_token=..."}
```

You should then visit the returned url with a browser or redirect your website visitor to this address. There, twitter should ask you to authorize the `Savonet Flows` application for the twitter user that you or your visitor are using.

Once done, you should be automatically redirected to `:link` where `:link` is the parameter passed in the initial `HTTP GET` request. From this point, all metadata updates for the that radio should be forwarded to the twitter account that was just authorized.

28.8 Command-line client

A command-line client is also available. It can be used to:

- Change your radio's name
- Change your radio's location
- Register and unregister twitter notifications

The code is still being developped. It can be accessed at this url: https://github.com/savonet/flows-client

CHAPTER 29

Fréquence 3

Fréquence 3 uses Liquidsoap mainly on the backstage, for different purposes:

- transcoding for different formats (OGG, weird MP3 relays. . .)
- scheduling and playlist for audio backup streams, and test streams
- blank detection

They look forward to using Liquidsoap even more, and work with the Savonet team to make sure this tool can ease the work of webradios :)

They provide an MP3 stream here.

CHAPTER 30

Geek Radio

The historical webradio, founded by David Baelde and Samuel Mimram at the ENS Lyon.

The very first version was, as many other radios, a Perl function called by Ices. It played files, one by one. On the campus, there was plenty of audio files available, so they soon wanted to index them and be able to ask easily for one file to be streamed. Samuel made a dirty campus indexer in OCaml, and David made an ugly Perl hack for adding user requests to the original system. It probably kind of worked for a while. Then they wanted something more, and realized it was all too ugly.

So they made the binding of libshout for OCaml and built the first streamer in pure OCaml. It had a simple telnet interface so an IRC bot could send user requests easily to it, same for the website. There were two request queues, one for users, one for admins. But it was still not so nicely designed, and they felt it when they needed more. They wanted scheduling, especially techno music at night.

Around that time students had to set up a project for one of their courses. David and Samuel proposed to build a complete flexible webradio system, that's Savonet. To give jobs to everybody, they had planned a complete rewriting of every part, with grand goals. A new website with so much features, a new intelligent multilingual bot, a new network libraries for glueing that, etc. Most died. But still, Liquidsoap was born, and they had plenty of new libraries for OCaml. Since then, Liquidsoap has been greatly enhanced, and is now spreading outside the ENS Lyon.

30.1 Features

The liquidsoap script schedules several static (but periodically reloaded) playlists played on different times, adds jingle to the usual stream every hour, adds short live interventions, or completely switches to live shows when available. It accepts user requests, which have priority over static playlists but not live shows, and adds speech-synthetized metadata information at the end of requests.

Geek Radio used to have a Strider daemon running to fill our database. Since that project is now dead, a simple hack is now used instead: bubble.

The usual way of sending a request is via an IRC bot, which queries the database and sends the chosen URI to liquidsoap.

CHAPTER 31

The Gstreamer encoder

The `%gstreamer` encoder can be used to encode streams using the `gstreamer` multimedia framework. This encoder extends liquidsoap with all available GStreamer formats (provided they are compatible with liquidsoap's model of streams, see Caveats section below), which includes a huge array of encoders.

31.1 Presentation

A basic understanding of gstreamer's pipelines and configuration should be expected in order to understand the following documentation.

The encoder's parameters are as follows:

```
%gstreamer(channels=2,
           audio="lamemp3enc",
           has_video=true,
           video="x264enc",
           muxer="mpegtsmux",
           metadata="metadata",
           log=5,
           pipeline="")
```

Without using the `pipeline` argument, the `audio` and `video` arguments are used to build the gstreamer pipeline used to encode. By setting the `log` parameter to a lower value or by using `set("log.level",..)`, you should be able to see some example.

31.2 Basic examples

Here are a couple of examples:

An MP3 encoder that expects sources of type `audio=2, video=0, midi=0`:

```
% liquidsoap 'output.file(%gstreamer(audio="lamemp3enc",
                                     muxer="",
                                     video="",
                                     log=3),...)'
(...)
2012/12/13 19:16:23 [encoder.gstreamer:3] Gstreamer encoder pipeline: appsrc
  name="audio_src" block=true caps="audio/x-raw,format=S16LE,layout=interleaved,
  channels=2,rate=44100" format=time ! queue ! audioconvert ! audioresample !
  lamemp3enc ! appsink name=sink sync=false emit-signals=true
```

A x264 encoder that expects sources of type `audio=0, video=1, midi=0`:

```
% liquidsoap 'output.file(%gstreamer(audio="",
                                     muxer="mpegtsmux",
                                     video="x264enc",
                                     log=3),...)'
(...)
2012/12/13 19:14:43 [encoder.gstreamer:3] Gstreamer encoder pipeline:  appsrc
  name="video_src" block=true caps="video/x-raw,format=RGBA,width=320,height=240,
  framerate=25/1,pixel-aspect-ratio=1/1" format=time blocksize=307200 ! queue !
  videoconvert ! videoscale add-borders=true ! videorate ! x264enc !
  mpegtsmux name=muxer ! appsink name=sink sync=false emit-signals=true
```

An MPEG TS encoder that expects sources of type `audio=2, video=1, midi=0`:

```
% liquidsoap 'output.file(%gstreamer(audio="lamemp3enc",
                                     muxer="mpegtsmux",
                                     video="x264enc",
                                     log=3),...)'
(...)
2012/12/13 19:18:09 [encoder.gstreamer:3] Gstreamer encoder pipeline: appsrc
  name="audio_src" block=true caps="audio/x-raw,format=S16LE,
  layout=interleaved,channels=2,rate=44100" format=time ! queue ! audioconvert
  ! audioresample ! lamemp3enc ! muxer. appsrc name="video_src" block=true
  caps="video/x-raw,format=RGBA,width=320,height=240,framerate=25/1,
  pixel-aspect-ratio=1/1" format=time blocksize=307200 ! queue ! videoconvert
  ! videoscale add-borders=true ! videorate ! x264enc ! muxer. mpegtsmux
  name=muxer ! appsink name=sink sync=false emit-signals=true
```

An ogg/vorbis+theora encoder that expects source of type `audio=1, video=1, midi=0`:

```
% liquidsoap 'output.file(%gstreamer(audio="vorbisenc",
                                     muxer="oggmux",
                                     video="theoraenc",
                                     channels=1,
                                     log=3),...)'
(...)
2012/12/13 19:21:17 [encoder.gstreamer:3] Gstreamer encoder pipeline: appsrc
  name="audio_src" block=true caps="audio/x-raw,format=S16LE,layout=interleaved,
  channels=1,rate=44100" format=time ! queue ! audioconvert ! audioresample !
  vorbisenc ! muxer. appsrc name="video_src" block=true caps="video/x-raw,
  format=RGBA,width=320,height=240,framerate=25/1,pixel-aspect-ratio=1/1"
  format=time blocksize=307200 ! queue ! videoconvert ! videoscale add-borders=true
  ! videorate ! theoraenc ! muxer. oggmux name=muxer ! appsink name=sink
  sync=false emit-signals=true
```

For advanced users, the `pipeline` argument can be used to override the whole pipeline. For instance:

```
% liquidsoap 'output.file(%gstreamer(pipeline="appsrc name=\"audio_src\"
    block=true caps=\"audio/x-raw,format=S16LE,layout=interleaved,
    channels=1,rate=44100\" format=time ! lamemp3enc ! appsink name=sink
    sync=false emit-signals=true",channels=1,log=3),...)'
(...)
```

31.3 Content type inference

When starting its sources and outputs, liquidsoap determines the content type of each source (audio, video and midi channels). During that process, encoders have to inform liquidsoap what type of sources they are expecting. It works as follows for the `%gstreamer` encoder:

- If the `audio` parameter is a string different than `""` then the encoder expects a stream with `channels` audio channels.
- If the `video` parameter is a string different than `""` then the encoder expects a stream with `1` video channel.
- If the `pipeline` parameter is a string different than `""` then the encoder expects a stream with `channels` audio channels and a video channels only if `has_video` is true.

The `has_video` parameter is only used when using the `pipeline` parameter.

31.4 Metadata

The `%gstreamer` encoder tries to also encode metadata attached to the stream. This requires that you specify a pipeline element named according to the `metadata` parameter (default: `"metadata"`) that can be used with GStreamer's `tag_setter` API. Here are two such examples:

An ogg/vorbis encoder with vorbis tags:

```
% liquidsoap 'output.file(%gstreamer(audio="vorbisenc ! vorbistag name='metadata'",
                                     muxer="oggmux",
                                     video=""),...)'
```

An MP3 encoder with id3v2 tags:

```
% liquidsoap 'output.file(%gstreamer(audio="lamemp3enc",
                                     muxer="id3v2mux",
                                     video="",
                                     metadata="muxer"),...)'
```

In the last example, we tell the `%gstreamer` encoder that the element for injecting metadata is named `"muxer"` because, for id3v2 tags, the gstreamer muxer element is also the element used to inject metadata and the `"muxer"` name is implicitely added by liquidsoap to the muxer element. You can see that by printing out the constructed pipeline, as shown before.

31.5 Caveats

When using the `%gstreamer` encoder, one must think of it as an encoder for an infinite stream. This, in particular, means that not all containers (muxers) will work. For instance, the AVI and MP4 containers need to write in their header informations that are only known with finite streams, such as the stream total's time and etc.. These containers are usually not fit for streaming, which is liquidsoap's main functionality.

CHAPTER 32

Harbor as HTTP server

The harbor server can be used as a HTTP server. You can use the function `harbor.http.register` to register HTTP handlers. Its parameters are are follow:

```
harbor.http.register(port=8080,method="GET",uri,handler)
```

where:

- `port` is the port where to receive incoming connections
- `method` is for the http method (or verb), one of: `"GET"`, `"PUT"`, `"POST"`, `"DELETE"`, `"OPTIONS"` and `"HEAD"`
- `uri` is used to match requested uri. Perl regular expressions are accepted.
- `handler` is the function used to process requests.

`handler` function has type:

```
(~protocol:string, ~data:string,
 ~headers:[(string*string)], string)->'a))->unit
where 'a is either string or ()->string
```

where:

- `protocol` is the HTTP protocol used by the client. Currently, one of `"HTTP/1.0"` or `"HTTP/1.1"`
- `data` is the data passed during a POST request
- `headers` is the list of HTTP headers sent by the client
- `string` is the (unparsed) uri requested by the client, e.g.: `"/foo?var=bar"`

The `handler` function returns HTTP and HTML data to be sent to the client, for instance:

```
HTTP/1.1 200 OK\r\n\
Content-type: text/html\r\n\
Content-Length: 35\r\n\
```

(continues on next page)

(continued from previous page)

```
\r\n\
<html><body>It works!</body></html>
```

(`\r\n` should always be used for line return in HTTP content)

The handler is a *string getter*, which means that it can be of either type `string` or type `()->string`. The former is used to returned the response in one call while the later can be used to returned bigger response without having to load the whole response string in memory, for instane in the case of a file.

For convenience, two functions, `http_response` and `http_response_stream` are provided to create a HTTP response string. `http_response` has the following type:

```
(?protocol:string,?code:int,?headers:[(string*string)],
 ?data:string)->string
```

where:

- `protocol` is the HTTP protocol of the response (default `HTTP/1.1`)
- `code` is the response code (default `200`)
- `headers` is the response headers. It defaults to `[]` but an appropriate `"Content-Length"` header is added if not set by the user and `data` is not empty.
- `data` is an optional response data (default `""`)

`http_response_stream` has the following type:

```
(?protocol:string,?code:int,?headers:[(string*string)],
 data_len:int,data:()->string)->string
```

where:

- `protocol` is the HTTP protocol of the response (default `HTTP/1.1`)
- `code` is the response code (default `200`)
- `headers` is the response headers. It defaults to `[]` but an appropriate `"Content-Length"` header is added if not set by the user and `data` is not empty.
- `data_len` is the length of the streamed response
- `data` is the response stream

Thess functions can be used to create your own HTTP interface. Some examples are:

32.1 Redirect Icecast's pages

Some source clients using the harbor may also request pages that are served by an icecast server, for instance listeners statistics. In this case, you can register the following handler:

```
# Redirect all files other
# than /admin.* to icecast,
# located at localhost:8000
def redirect_icecast(~protocol,~data,~headers,uri) =
   http_response(
     protocol=protocol,
     code=301,
```

(continues on next page)

(continued from previous page)

```
     headers=[("Location","http://localhost:8000#{uri}")]
   )
end

# Register this handler at port 8005
# (provided harbor sources are also served
#  from this port).
harbor.http.register(port=8005,method="GET",
                     "^/(?!admin)",
                     redirect_icecast)
```

Another alternative, less recommended, is to directly fetch the page's content from the Icecast server:

```
# Serve all files other
# than /admin.* by fetching data
# from Icecast, located at localhost:8000
def proxy_icecast(~protocol,~data,~headers,uri) =
  def f(x) =
    # Replace Host
    if string.capitalize(fst(x)) == "HOST" then
      "Host: localhost:8000"
    else
      "#{fst(x)}: #{snd(x)}"
    end
  end
  headers = list.map(f,headers)
  headers = string.concat(separator="\r\n",headers)
  request =
    "#{method} #{uri} #{protocol}\r\n\
     #{headers}\r\n\r\n"
  get_process_output("echo #{quote(request)} | \
                      nc localhost 8000")
end

# Register this handler at port 8005
# (provided harbor sources are also served
#  from this port).
harbor.http.register(port=8005,method="GET",
                     "^/(?!admin)",
                     proxy_icecast)
```

This method is not recommended because some servers may not close the socket after serving a request, causing `nc` and liquidsoap to hang.

32.2 Get metadata

You can use harbor to register HTTP services to fecth/set the metadata of a source. For instance, using the JSON export function `json_of`:

```
meta = ref []

# s = some source

# Update current metadata
```

(continues on next page)

(continued from previous page)

```
# converted in UTF8
def update_meta(m) =
  m = metadata.export(m)
  recode = string.recode(out_enc="UTF-8")
  def f(x) =
    (recode(fst(x)),recode(snd(x)))
  end
  meta := list.map(f,m)
end

# Apply update_metadata
# every time we see a new
# metadata
s = on_metadata(update_meta,s)

# Return the json content
# of meta
def get_meta(~protocol,~data,~headers,uri) =
  m = !meta
  http_response(
    protocol=protocol,
    code=200,
    headers=[("Content-Type","application/json; charset=utf-8")],
    data=json_of(m)
  )
end

# Register get_meta at port 700
harbor.http.register(port=7000,method="GET","/getmeta",get_meta)
```

Once the script is running, a GET/POST request for `/getmeta` at port `7000` returns the following:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{
  "genre": "Soul",
  "album": "The Complete Stax-Volt Singles: 1959-1968 (Disc 8)",
  "artist": "Astors",
  "title": "Daddy Didn't Tell Me"
}
```

Which can be used with AJAX-based backends to fetch the current metadata of source `s`

32.3 Set metadata

Using `insert_metadata`, you can register a GET handler that updates the metadata of a given source. For instance:

```
# s = some source

# x is of type ((metadata)->unit)*source
# first part is a function used to update
# metadata and second part is the source
# whose metadata are updated
```

(continues on next page)

(continued from previous page)

```
x = insert_metadata(s)

# Get the function
insert = fst(x)

# Redefine s as the new source
s = snd(x)

# The handler
def set_meta(~protocol,~data,~headers,uri) =
  # Split uri of the form request?foo=bar&...
  # into (request,[("foo","bar"),..])
  x = url.split(uri)

  # Filter out unusual metadata
  meta = metadata.export(snd(x))

  # Grab the returned message
  ret =
    if meta != [] then
      insert(meta)
      "OK!"
    else
      "No metadata to add!"
  end

  # Return response
  http_response(
   protocol=protocol,
   code=200,
   headers=[("Content-Type","text/html")],
   data="<html><body><b>#{ret}</b></body></html>"
  )
end

# Register handler on port 700
harbor.http.register(port=7000,method="GET","/setmeta",set_meta)
```

Now, a request of the form `http://server:7000/setmeta?title=foo` will update the metadata of source `s` with `[("title","foo")]`. You can use this handler, for instance, in a custom HTML form.

CHAPTER 33

Limitations

When using harbor's HTTP server, please be warned that the server is **not** meant to be used under heavy load. Therefore, it should **not** be exposed to your users/listeners if you expect many of them. In this case, you should use it as a backend/middle-end and have some kind of caching between harbor and the final user. In particular, the harbor server is not meant to server big files because it loads their entire content in memory before sending them. However, the harbor HTTP server is fully equipped to serve any kind of CGI script.

CHAPTER 34

Harbor input

Liquidsoap is also able to receive a source using icecast or shoutcast source protocol with the `input.harbor` operator. Using this operator, the running liquidsoap will open a network socket and wait for an incoming connection.

This operator is very useful to seamlessly add live streams into your final streams: you configure the live source client to connect directly to liquidsoap, and manage the switch to and from the live inside your script.

Additionally, liquidsoap can handle many simulataneous harbor sources on different ports, with finer-grained authentication schemes that can be particularly useful when used with source clients designed for the shoutcast servers.

SSL support in harbor can be enabled using of of the following `opam` packages: `ssl`, `osx-secure-transport`. If enabled using `ssl`, `input.harbor.ssl` will be available. If enabled with `osx-secure-transport`, it will be `input.harbor.secure_transport`.

34.1 Parameters

The global parameters for harbor can be retreived using `liquidsoap --conf-descr-key harbor`. They are:

- `harbor.bind_addr`: IP address on which the HTTP stream receiver should listen. The default is `"0.0.0.0"`. You can use this parameter to restrict connections only to your LAN.
- `harbor.timeout`: Timeout for source connection, in seconds. Defaults to `30.`.
- `harbor.verbose`: Print password used by source clients in logs, for debugging purposes. Defaults to: `false`
- `harbor.reverse_dns`: Perform reverse DNS lookup to get the client's hostname from its IP. Defaults to: `true`
- `harbor.icy_formats`: Content-type (mime) of formats which allow shout (ICY) metadata update. Defaults to: `["audio/mpeg"; "audio/aacp"; "audio/aac"; "audio/x-aac"; "audio/wav"; "audio/wave"]`

If SSL support was enabled via `ssl`, you will have the following additional settings:

- `harbor.ssl.certificate`: Path to the SSL certificate.

- `harbor.ssl.private_key`: Path to the SSL private key (openssl only).
- `harbor.ssl.password`: Optional password to unlock the private key.

Obtaining a proper SSL certificate can be tricky. You may want to start with a self-signed certificate first. You can obtain a free, valid certificate at: https://letsencrypt.org/

If SSL support is enable via `osx-secure-transport`, you will have the same settings but named: `harbor.secure_transport.*`.

To create a self-signed certificate for local testing you can use the following one-liner:

```
openssl req -x509 -newkey rsa:4096 -sha256 -nodes -keyout server.key -out server.crt -
↪subj "/CN=localhost" -days 3650
```

You also have per-source parameters. You can retreive them using the command `liquidsoap -h input.harbor`. The most important one are:

- `user`, `password`: set a permanent login and password for this harbor source.
- `auth`: Authenticate the user according to a specific function.
- `port`: Use a custom port for this input.
- `icy`: Enable ICY (shoutcast) source connections.
- `id`: The mountpoint registered for the source is also the id of the source.

When using different ports with different harbor inputs, mountpoints are attributed per-port. Hence, there can be a harbor input with mountpoint `"foo"` on port `1356` and a harbor input with mountpoint `"foo"` on port `3567`. Additionaly, if an harbor source uses custom port `n` with shoutcast (ICY) source protocol enabled, shoutcast source clients should set their connection port to `n+1`.

The `auth` function is a function, that takes a pair `(user,password)` and returns a boolean representing whether the user should be granted access or not. Typical example can be:

```
def auth(user,password) =
  # Call an external process to check
  # the credentials:
  # The script will return the string
  # "true" of "false"
  #
  # First call the script
  ret = get_process_lines("/path/to/script \
         --user=#{user} --password=#{password}")
  # Then get the first line of its output
  ret = list.hd(default="",ret)
  # Finally returns the boolean represented
  # by the output (bool_of_string can also
  # be used)
  if ret == "true" then
    true
  else
    false
  end
end
```

In the case of the `ICY` (shoutcast) source protocol, there is no `user` parameter for the source connection. Thus, the user used will be the `user` parameter passed to the `input.harbor` source.

When using a custom authentication function, in case of a `ICY` (shoutcast) connection, the function will receive this value for the username.

34.2 Usage

When using harbor inputs, you first set the required settings, as described above. Then, you define each source using `input.harbor("mountpoint")`. This source is faillible and will become available when a source client is connected.

The unlabeled parameter is the mount point that the source client may connect to. It should be `"/"` for shoutcast source clients.

The source client may use any of the recognized audio input codec. Hence, when using shoucast source clients, you need to have compiled liquidsoap with mp3 decoding support (`ocaml-mad`)

A sample code can be:

```
set("harbor.bind_addr","0.0.0.0")

# Some code...

# This defines a source waiting on mount point
# /test-harbor
live = input.harbor("test-harbor",port=8080,password="xxx")

# This is the final stream.
# Uses the live source as soon as available,
# and don't wait for an end of track, since
# we don't want to cut the beginning of the live
# stream.
#
# You may insert a jingle transition here...
radio = fallback(track_sensitive=false,
                 [live,files])
```

CHAPTER 35

Get help

Liquidsoap is a self-documented application, which means that it can provide help about several of its aspects. You will learn here how to get help by yourself, by asking liquidsoap. If you do not succeed in asking the tool, you can of course get help from humans, preferably on the mailing list `savonet-users@lists.sf.net`.

35.1 Scripting API

When scripting in liquidsoap, one uses functions that are either *builtin* (*e.g.* `fallback` or `output.icecast`) or defined in the script library (*e.g* `out`). All these functions come with a documentation, that you can access by executing `liquidsoap -h FUNCTION` on the command-line. For example:

```
$ liquidsoap -h sine
*** One entry in scripting values:
Generate a sine wave.
Category: Source / Input
Type: (?id:string, ?duration:float, ?float)->source
Parameters:
* id :: string (default "")
    Force the value of the source ID.
* duration :: float (default 0.)
* (unlabeled) :: float (default 440.)
    Frequency of the sine.
```

Of course if you do not know what function you need, you'd better go through the API reference.

35.2 Server commands

The server (*cf.* the server tutorial) offers some help about its commands. Once connected (either via a TCP or UNIX socket) the `help` command gives you a list of available commands together with a short usage line. You can then get more detailed information about a specific command by typing `help COMMAND`:

```
$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
help
Available commands:
[...]
| queue.ignore <rid>
| queue.push <uri>
| queue.queue
[...]
Type "help <command>" for more information.
END
help queue.push

Help for command queue.push.

Usage: queue.push <uri>
  Push a new request in the queue.
END
```

35.3 Settings

Liquidsoap scripts contain expression like `set("log.stdout",true)`. These are *settings*, global variables affecting the behaviour of the application. Here, the first parameter identifies a setting its path, and the second one specifies its new value.

You can have a list of available settings, with their documentation, by running `liquidsoap --conf-descr`. If you are interested in a particular settings section, for example server-related stuff, use `liquidsoap --conf-descr-key server`.

The output of these commands is a valid liquidsoap script, which you can edit to set the values that you want, and load it (implicitly or not) before you other scripts.

You can browse online the list of available settings.

35.4 All plugins

Several aspects of liquidsoap work with a notion of plugin: builtin scripting functions, audio decoders for files and streams, metadata decoders, protocols, etc. The list of plugins can be used to check that your build of liquidsoap has such or such feature, or simply to browse available functions – actually, the reference is built from that output.

You can get the pretty hairy list of all available plugins from the command `liquidsoap --list-plugins`, or `liquidsoap --list-plugins-xml` for a more parsable XML output.

CHAPTER 36

HTTP input

Liquidsoap can create a source that pulls its data from an HTTP location. This location can be a distant file or playlist, or an icecast or shoutcast stream.

To use it in your script, simply create a source that way:

```
# url is a HTTP location, like
# http://radiopi.org:8080/reggae
source = input.http(url)
```

This operator will pull regulary the given location for its data, so it should be used for locations that are assumed to be available most of the time. If not, it might generate unnecessary traffic and polute the logs. In this case, it is perhaps better to inverse the paradigm and use the input.harbor operator.

CHAPTER 37

ICY metadata

ICY metadata is the name for the mechanism used to update metadata in icecast's source streams. The techniques is primarily intended for data formats that do not support in-stream metadata, such as mp3 or AAC. However, it appears that icecast also supports ICY metadata update for ogg/vorbis streams.

When using the ICY metadata update mechanism, new metadata are submitted separately from the stream's data, via a http GET request. The format of the request depends on the protocol you are using (ICY for shoutcast and icecast 1 or HTTP for icecast 2).

Starting with 1.0, you can do several interesting things with icy metadata updates in liquidsoap. We list some of those here.

37.1 Enable/disable ICY metadata updates

You can enable or disable icy metadata update in `output.icecast` by setting the `icy_metadata` parameter to either `"true"` or `"false"`. The default value is `"guess"` and does the following:

- Set `"true"` for: mp3, aac, aac+, wav
- Set `"false"` for any format using the ogg container

You may, for instance, enable icy metadata update for ogg/vorbis streams.

37.2 Update metadata manually

The function `icy.update_metadata` implements a manual metadata update using the ICY mechanism. It can be used independently from the `icy_metadata` parameter described above, provided icecast supports ICY metadata for the intended stream.

For instance the following script registers a telnet command name `metadata.update` that can be used to manually update metadata:

```
def icy_update(v) =
  # Parse the argument
  l = string.split(separator=",",v)
  def split(l,v) =
    v = string.split(separator="=",v)
    if list.length(v) >= 2 then
      list.append(l,[(list.nth(v,0,default=""),list.nth(v,1,default=""))])
    else
      l
    end
  end
  meta = list.fold(split,[],l)

  # Update metadata
  icy.update_metadata(mount="/mystream",password="hackme",
                      host="myserver.net",meta)
  "Done !"
end

server.register("update",namespace="metadata",
                description="Update metadata",
                usage="update title=foo,album=bar,..",
                icy_update)
```

As usual, `liquidsoap -h icy.update_metadata` lists all the arguments of the function.

CHAPTER 38

Using in production

The full installation of liquidsoap will typically install `/etc/liquidsoap`, `/etc/init.d/liquidsoap` and `/var/log/liquidsoap`. All these are meant for a particular usage of liquidsoap when running a stable radio.

Your production `.liq` files should go in `/etc/liquidsoap`. You'll then start/stop them using the init script, *e.g.* `/etc/init.d/liquidsoap start`. Your scripts don't need to have the `#!` line, and liquidsoap will automatically be ran on daemon mode (`-d` option) for them.

You should not override the `log.file.path` setting because a logrotate configuration is also installed so that log files in the standard directory are truncated and compressed if they grow too big.

It is not very convenient to detect errors when using the init script. We advise users to check their scripts after modification (use

```
liquidsoap --check /etc/liquidsoap/script.liq```
)
before effectively restarting the daemon.
```

CHAPTER 39

Installing Savonet/Liquidsoap

Note These instructions are from the documentation from liquidsoap `1.4.2`. Make sure to consult the instructions from the version you wish to install, most likely the latest stable release.

You can install liquidsoap with OPAM (recommended) or from source, or using a package available for your distribution (not covered by this documentation).

- *Using OPAM*
- *Debian/Ubuntu*
- *Windows*
- *From source*
- *Latest development version*

39.1 Install using OPAM

The recommended method to install liquidsoap is by using the OCaml Package Manager. OPAM is available in all major distributions and on windows. We actively support the liquidsoap packages there and its dependencies. You can read here about how to use OPAM. In order to use it:

- you should have at least OPAM version 2.0,
- you should have at least OCaml version 4.08.0, which can be achieved by typing

```
opam switch create 4.08.0
```

A typical installation with MP3 and Vorbis encoding/decoding and icecast support is done by executing:

```
opam depext taglib mad lame vorbis cry samplerate liquidsoap
opam install taglib mad lame vorbis cry samplerate liquidsoap
```

- `opam depext ...` takes care of installing the required external dependencies. In some cases external dependencies might be missing for your system. If that is the case, please report it to us!

- Finally `opam install ...` installs the packages themselves.

Most of liquidsoap's dependencies are only optionally installed by OPAM. For instance, if you want to enable opus encoding and decoding after you've already installed liquidsoap, you should execute the following:

```
opam depext opus
opam install opus
```

`opam info liquidsoap` should give you the list of all optional dependencies that you may enable in liquidsoap.

If you need to run liquidsoap as daemon, we provide a package named `liquidsoap-daemon`. See savonet/liquidsoap-daemon for more information.

You can also install liquidsoap or any of its dependencies from source using OPAM. For instance:

```
git clone https://github.com/savonet/liquidsoap.git
cd liquidsoap
opam pin add liquidsoap .
```

Most dependencies should be compatible with OPAM pinning. Let us know if you find one that isn't.

39.2 Debian/Ubuntu

We generate debian and ubuntu packages automatically as part of our CI workflow. These packages are available for quick testing of `liquidsoap` on certain Debian and Ubuntu distributions. However, we do not recommend them yet for production purposes.

Please note We cannot guarantee that any of the distribution below will remain available at all time and we reserve the right to purge old versions of the packages at any time. If you plan on using some of these packages for any sort of production use, make sure to copy them and use your own distribution channels.

Here's how to install:

- First install the repository signing key:

```
[sudo] apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 20D63CCDDD0F62C2
```

- Then one of the following source:

debian/stable:

```
[sudo] echo deb http://deb.liquidsoap.info/debian stable main >> /etc/apt/sources.
↪list.d/liquidsoap.list
```

debian/testing:

```
[sudo] echo deb http://deb.liquidsoap.info/debian testing main >> /etc/apt/sources.
↪list.d/liquidsoap.list
```

debian/stretch:

```
[sudo] echo deb http://deb.liquidsoap.info/debian stretch main >> /etc/apt/sources.
↪list.d/liquidsoap.list
```

ubuntu/bionic:

```
[sudo] add-apt-repository ppa:sergey-dryabzhinsky/ffmpeg
[sudo] echo deb http://deb.liquidsoap.info/ubuntu bionic main >> /etc/apt/sources.
↪list.d/liquidsoap.list
```

(continues on next page)

(continued from previous page)

ubuntu/disco:

```
[sudo] echo deb http://deb.liquidsoap.info/ubuntu disco main >> /etc/apt/sources.list.
↪d/liquidsoap.list
```

You can now see the list of available packages:

```
apt-cache show liquidsoap
```

Package names are of the form: `liquidsoap-<commit>` and `liquidsoap-<branch>`. For instance, to install the latest `master` you can do:

```
[sudo] apt-get install liquidsoap-master
```

39.3 Windows

You can download a liquidsoap for windows from our release page, starting with version `1.3.4`.

Liquidsoap for windows is built using opam-cross. The build process is documented in our docker files. `Dockerfile.win32-deps` installs all the mxe dependencies and `Dockerfile.win32` produces the actual liquidsoap binary.

You might want to refer to each project, mxe and opam-cross for more details about cross-compiling for windows.

39.4 Installing from source

You can download source code published by Savonet from the github releases page.

The recommended way for newcomers is to use the liquidsoap-full-xxx.tar.gz tarball. This tarball includes all required OCaml bindings and allows you to compile and install liquidsoap in a single `configure`, `make` and `make install` procedure. You will still need the corresponding C libraries and their development files, though.

You will then have to build the source.

39.5 Latest development version

If you want a cutting-edge version, you can use the git repository. To get a copy of it, you have to run:

```
git clone https://github.com/savonet/liquidsoap-full.git liquidsoap
cd liquidsoap
make init
```

After, that you have to create a list of modules that you want to compile. A good starting point is to do

```
cp PACKAGES.minimal PACKAGES
```

and edit the `PACKAGES` file to uncomment the libraries you are interested in. You should then run the configuration scripts by

```
./bootstrap
./configure
```

and finally build Liquidsoap:

```
make
```

After that, you should synchronize the repository from time to time using

```
make update
```

Some more explanations can be found in the build instructions.

CHAPTER 40

Exporting values using JSON

Liquidsoap can export any language value in JSON using `json_of`.

The format is the following :

- `() : unit -> null`
- `true: bool -> true`
- `"abc" : string -> "abc"`
- `23 : int -> 23`
- `2.0 : float -> 2.0`
- `[2,3,4] : [int] -> [2,3,4]`
- `[("f",1),("b",4)] : [(string*int)] -> { "f": 1, "b": 4 }`
- `("foo",123) : string*int -> [ "foo", 123 ]`
- `s : source -> "<source>"`
- `r : ref(int) -> { "reference":4 }`
- `%mp3 : format(...) -> "%mp3(stereo,bitrate=128,samplerate=44100)"`
- `r : request(...) -> "<request>"`
- `f : (...)->_ -> "<fun>"`

The two particular cases are:

- Products are exported as lists.
- Lists of type `[(string*'a)]` are exported as objects of the form `{"key": value}`.

Output format is pretty printed by default. A compact output can be obtained by using the optional argument: `compact=true`.

CHAPTER 41

Importing values using JSON

If compiled with `yojson` support, Liquidsoap can also parse JSON data into values. using `of_json`.

The format is a subset of the format of exported values with the notable difference that only ground types (`int`, `floats`, `string`, ...) are supported and not variable references, sources, formats, requests and functions:

- `null -> () : unit`
- `true/false -> true/false : bool`
- `"abc" -> "abc" : string`
- `23 -> 23 : int`
- `2.0 -> 2.0 : float`
- `[2,3,4] -> [2,3,4] : int`
- `{"f": 1, "b": 4} -> [("f",1),("b",4)] : [(string*int)]`
- `[ "foo", 123 ] -> ("foo",123) : string*int`

The JSON standards specify that a proper JSON payload can only be an array or an object. However, simple integers, floats, strings and null values are also accepted by Liquidsoap.

The function `of_json` has the following type:

```
  (default:'a,string)->'a
```

The default parameter is very important in order to assure type inference of the parsed value. Its value constrains the parser to only recognize JSON data of the the default value's type and is returned in case parsing fails.

Suppose that we want to receive a list of metadata, encoded as an object:

```
{ "title": "foo",
 "artist": "bar" }
```

Then, you would use of_json with default value `[("error","fail")]` and do:

```
# Parse metadata from json
m = of_json(default= [("error","fail")], json_string)
```

The type of the default value constrains the parser. For instance, in the above example, a JSON string `"[1,2,3,4]"` will not be accepted and the function will return the values passed as default.

You can use the default value in two different ways:

- To detect that the received json string was invalid/could not be parsed to the expected type. In the example above, if `of_json` return a metadata value of `[("error","fail")]` (the default) then you can detect in your code that parsing has failed.
- As a default value for the rest of the script, if you do not want to care about parsing errors.. This can be useful for instance for JSON-RPC notifications, which should not send any response to the client anyway.

If your JSON object is of mixed type, like this one:

```
{ "uri": "https://...",
  "metadata": { "title": "foo", "artist": "bar" } }
```

You can parse it in multiple steps. For instance:

```
# First parse key,value list:
hint = [("key","value")]
data = of_json(default=hint,payload)
print(data["uri"]) # "https://..."

# Then key -> (key,value) list
hint = [("list",[("key","value")])]
data = of_json(default=hint,payload)
m    = list.assoc(default=[],"metadata",data)
print(m["title"]) # "foo"
```

CHAPTER 42

LADSPA plugins in Liquidsoap

LADSPA is a standard that allows software audio processors and effects to be plugged into a wide range of audio synthesis and recording packages.

If enabled, Liquidsoap supports LADSPA plugins. In this case, installed plugins are detected at run-time and are all available in Liquidsoap under a name of the form: `ladspa.plugin`, for instance `ladspa.karaoke`, `ladspa.flanger` etc..

The full list of those operators can be found using `liquidsoap --list-plugins`. Also, as usual, `liquidsoap -h ladspa.plugin` returns a detailed description of each LADSPA's operators. For instance:

```
./liquidsoap -h ladspa.flanger
*** One entry in scripting values:
Flanger by Steve Harris <steve@plugin.org.uk>.
Category: Source / Sound Processing
Type: (?id:string,?delay_base:'a,?feedback:'b,
 ?lfo_frequency:'c,?max_slowdown:'d,
 source(audio='#e,video='#f,midi='#g))->
source(audio='#e,video='#f,midi='#g)
where 'a, 'b, 'c, 'd is either float or ()->float
Flag: hidden
Parameters:
* id : string (default "")
    Force the value of the source ID.
* delay_base : anything that is either float or ()->float (default 6.32499980927)
    Delay base (ms) (0.1 <= delay_base <= 25).
* feedback : anything that is either float or ()->float (default 0.)
    Feedback (-1 <= feedback <= 1).
* lfo_frequency : anything that is either float or ()->float (default 0.334370166063)
    LFO frequency (Hz) (0.05 <= lfo_frequency <= 100).
* max_slowdown : anything that is either float or ()->float (default 2.5)
    Max slowdown (ms) (0 <= max_slowdown <= 10).
* (unlabeled) : source(audio='#e,video='#f,midi='#g) (default None)
```

For advanced users, it is worth nothing that most of the parameters associated with LADSPA operators can take a function, for instance in the above: `max_slowdown : anything that is either float or ()->float`

. This means that those parameters may be dynamically changed while running a liquidsoap script.

CHAPTER 43

Liquidsoap's scripting language

Liquidsoap's scripting language is a simple functional language, with labels and optional parameters. It is statically typed, but infers types – you don't have to write any types. It allows the direct handling of liquidsoap notions such as sources and requests, and also provides a convenient syntax for specifying time intervals.

The language's parser expects UTF8 as input so you should make sure that your scripts are written in this encoding, in particular if you include strings or variable names with non-ASCII characters in them.

43.1 Constants

The constants and their syntax are quite common:

- integers, such as `42`;
- floats, such as `3.14`;
- booleans `true` and `false`;
- strings, such as `"foo"` or `'bar'`.

Beware: `3.0` is not an integer and `5` is not a float, the dot matters.

Strings might be surrounded by double or single quotes. In both cases, you can escape the quote you're using: `"He said: \"Hello, you\"."` is valid but `'He said: "Hello, you".'` is equivalent and nicer.

You can include variables in a string using the `#{...}` syntax: `"foo #{quote(my_var)} bar"` is equivalent to `"foo " ^ quote(my_var) ^ " bar"`.

Finally, strings can be interpolated using the following syntax:

```
# s = 'This is an $(name) string. \
       This is a $(if $(value),"$(value)","undefined") value.';;
# s % [("name","interpolated")];;
- : string = "This is an interpolated string.This is a undefined value."
# s % [("name","interpolated"),("value","defined")];;
- : string = "This is an interpolated string.This is a defined value."
```

Most notably, `output.file` can use string interpolation to specify a different file name using the source's metadata.

43.2 Expressions

You can form expressions by using:

- Constants and variable identifiers. Identifiers start with an alphabetic character or an underscore, followed by alphanumerics, underscores, dots and quotes: `[A-Z a-z _][A-Z a-z 0-9 _.']*`.
- Lists and pairs: `[expr,expr,...]` and `(expr,expr)`.
- Comparison of values is done using `expr == expr` and its negation is `expr != expr`. Most other usual operations are available, allowing usual things like `1+1 < 11`.
- Application `f(x,y)` of arguments to a function. Application of labeled parameters is as follows: `f(x, foo=1,y,bar="baz")`. The interest of labels is that the order of two parameters doesn't matter as long as they have different labels.
- Anonymous functions: `fun (arglist) -> expr`. Some arguments might have a label or an optional value. For example, the definition of a function with two named parameters, the second one being optional with default value `13` is as follows: `fun (~foo,~bar=13) -> ...`.
- Definitions using def-end: `def pi = 3.14 end` defining a ground value, `def source(x) = wrap2(wrap1(x)) end` defining a function. The = is optional, you may prefer multi-line definitions without it. The arguments of a defined function are specified in the same way as for anonymous functions.
- Shorter definitions using the equality: `pi = 3.14`. This is never an assignment, only a new local definition!
- Conditionals `if expr then expr else expr end`, or more generally `if expr then expr (elsif expr then expr)* (else expr)? end`. The `else` block can be omitted if the purpose of the conditional is not to compute a value (*e.g.* an integer or a list of strings) but only to have a side effect (*e.g.* printing something in one case, not doing anything in the other).
- Sequencing: expressions may be sequenced, just juxtapose them. Usually one puts one expression per line. Optionally, they can be separated by a semicolon. The evaluation of a sequence triggers that of all of its sub-expressions, its value is that of the last sub-expression. Accordingly, the type of a sequence is that of its last sub-expression.
- Variable references are defined as: `reference = ref "some string"`. New values can be set via: `reference := "new value"`. The contents of a reference can be retrieved by `!reference`.
- Parenthesis can be used to delimit explicitly expressions. In some places where only expressions can be written, as opposed to sequences of expressions, the `begin .. end` block can be used to explicitly form a simple expression from a sequence. This notably happens with the simple form of definitions without `def .. end`, and in the body of anonymous functions. For example `fun (x) -> f1(x) ; f2(x)` will be read as `(fun (x) -> f1(x)) ; f2(x)` not as `fun (x) -> begin f1(x) ; f2(x) end`.
- Code blocks: `{ expr }` is a shortcut for `fun () -> expr`.

No assignment, only definitions. `x = expr` doesn't modify `x`, it just defines a new `x`. The expression `(x = s1 ; def y = x = s2 ; (x,s3) end ; (y,x))` evaluates to `((s2,s3),s1)`.

Function. The return value of a function is the evaluation of its body where parameters have been substituted by their values. Accordingly, the type of the body is the return type of the function. If the body is a sequence, the return value will thus be its last expression, and the return type its type.

```
def foo ()
  a = bar()
  b = 1
```

(continues on next page)

(continued from previous page)

```
  "string"
end
# The return type of foo is string.
# The full type of foo is ()->string.
```

Recursive functions can be defined using the `rec` keyword:

```
def rec fact(n) =
  if n == 1 then
    1
  else
    n * fact(n-1)
  end
end
```

Type of an application. The type of an application is the return type of function if all mandatory arguments are applied. With the function `foo` previously defined, `foo()` is a string. Otherwise, the application is ‘‘partial”, and the expression still has a function type.

Partial application. Application of arguments can be partial. For example if `f` takes two integer arguments, `f(3)` is the function waiting for the second argument. This can be useful to instantiate once for all dummy parameters of a function:

```
out = output.icecast(%vorbis, host="streamer",port="8080",
                             password="sesame")
# out is a function waiting for the other parameters
out(bitrate=112, my_radio)
```

Labels. Labeled and unlabeled parameters can be given at any place in an application. The order of the arguments is up to permutation of arguments of distinct labels. For example `f(x,foo=1)` is the same as `f(foo=1,x)`, both are valid for any function `f(x,~foo,...)`. It makes things easier for the user, and gives its full power to the partial application.

Optional arguments. Functions can be defined with an optional value for some parameter (as in `def f(x="bla", ~foo=1) = ... end`), in which case it is not required to apply any argument on the label `foo`. The evaluation of the function is triggered after the first application which instantiated all mandatory parameters.

43.3 Types

We believe in static typing especially for a script which is intended to run during weeks: we don’t want to notice a mistake only when the special code for your rare live events is triggered! Moreover, we find it important to show that even for a simple script language like that, it is worth implementing type inference. It’s not that hard, and makes life easier.

The basic types are `int`, `float`, `bool` and `string`. Corresponding to pairs and lists, you get `(T*T)` and `[T]` types – all elements of a list should have the same type. For example, `[(1,"un"),(2,"deux")]` has type `[(int*string)]`.

There are several types that are specific to liquidsoap, such as `source`, `request`, `format`. Those three types are parametrized by the kind of stream that they carry. This is described in more details in a dedicated page.

A function type is noted as `(arg_types) -> return_type`. Labeled arguments are denoted as `~label:T` or `?label:T` for optional arguments. For example the following function has type `(source,source,?jingle:string) -> source`.

```
fun (from,to,~jingle=default) ->
  add ([ sequence([single(jingle), fallback([])]),
         fade.in(to) ])
```

43.4 Time intervals

The scripting language also has a syntax extension for simply specifying time intervals.

A date can be specified as `_w_h_m_s` where `_` are integers. It has the following meaning:

- `w` stands for weekday, ranging from 0 to 7, where 1 is monday, and sunday is both 0 and 7.
- `h` stands for hours, ranging from 0 to 23.
- `m` stands for minutes, from 0 to 59.
- `s` stands for seconds, from 0 to 59.

All components `w`, `h`, `m` and `s` are optional. Finally, the `m` can be omitted in dates of the form `_h_` such as `12h30`.

It is possible to use 24 (resp. 60) as the upper bound for hours (resp. seconds or minutes) in an interval, for example in `12h-24h`.

Time intervals can be either of the form `DATE-DATE` or simply `DATE`. Their meaning should be intuitive: `10h-10h30` is valid everyday between 10:00 and 10:30; `0m` is valid during the first minute of every hour.

This is typically used for specifying switch predicates:

```
switch([
  ({ 20h-22h30 }, prime_time),
  ({ 1w }, monday_source),
  ({ (6w or 7w) and 0h-12h }, week_ends_mornings),
  ({ true }, default_source)
])
```

43.5 Includes

You can include other files, to compose complex configurations from multiple blocks of utility or configuration directives.

```
# Store passwords in another configuration file,
# so that the main config can be safely version-controlled.
%include "passwords.liq"

# Use the definitions from the other file here.
```

In the command `%include "file"` the path is relative to the script file. In `%include <file>`, it is relative to the library directory of Liquidsoap.

CHAPTER 44

Customize metadata using Liquidsoap

Liquidsoap has several mechanism for manipulating the metadata attached to your stream. In this page we quickly detail and compare the different operators, see the language reference for full details about them.

Warning. The protocol used by Shoutcast and Icecast before version 2 does not support many fields. It mainly support one: `song`. So, if you need to customize the metadata displayed by these servers, you should customize only the `song` metadata.

44.1 The annotate protocol

The metadata are read from files, so the most simple way is to properly tag the files. However, if it not possible to modify the files for some reason, the `annotate` protocol can be used in playlists to insert and modify some metadata. For instance, in the playlist

```
annotate:title="Title 1",artist="Artist 1":music1.mp3
annotate:title="Title 2",artist="Artist 2":music2.mp3
```

the title metadata for file music1.mp3 will be overridden and changed to ‘‘Title 1” (and similarly for the artist).

44.2 Map metadata

The `map_metadata` operator applies a specified function to transform each metadata chunk of a stream. It can be used to add or decorate metadata, but is also useful in more complex cases.

A simple example using it:

```
# A function applied to each metadata chunk
def append_title(m) =
  # Grab the current title
  title = m["title"]

```

(continues on next page)

(continued from previous page)

```
  # Return a new title metadata
  [("title","#{title} - www.station.com")]
end

# Apply map_metadata to s using append_title
s = map_metadata(append_title, s)
```

The effect of `map_metadata` by default is to update the metadata with the returned values. Hence in the function `append_title` defined in the code above returns a new metadata for the label `title` and the other metadata remain untouched. You can change this by using the `update` option, and you can also remove any metadata (even empty one) using the `strip` option.

See the documentation on `map_metadata` for more details.

44.3 Insert metadata

44.3.1 Using the telnet server

This operator is used for inserting metadata using a server command. If you have an `server.insert_metadata` node named `ID` in your configuration, as in

```
server.insert_metadata(id="ID", source)
```

you can connect to the server (either telnet or socket) and execute commands like

```
ID.insert key1="val1",key2="val2",...
```

44.3.2 In Liquidsoap

Sometimes it is desirable to change the metadata dynamically when an event occurs. In this case, the function `insert_metadata` (not to be confused with `server.insert_metadata` above) can be used: when applied to a source it returns a pair constituted of a function to update metadata and the source with inserted metadata.

For instance, suppose that you want to insert metadata on the stream using the OSC protocol. When a pair of strings `title''The new title"` is recieved on `/metadata`, we want to change the title of the stream accordingly. This can be achieved as follows.

```
# Our main music source
s = playlist("...")
s = mksafe(s)

# Create a function to insert metadata
ms = insert_metadata(s)
# The function to insert metadata
imeta = fst(ms)
# The source with inserted metadata
s = snd(ms)

# Handler for OSC events (gets pairs of strings)
def on_meta(m) =
  # Extract the label
  label = fst(m)
```

(continues on next page)

(continued from previous page)

```
  # Extract the value
  value = snd(m)
  # A debug message
  print("Insert metadata #{label} = #{value}")
  # Insert the metadata
  imeta([(label,value)])
end

# Call the above handler when we have a pair of strings on /metadata
osc.on_string_pair("/metadata",on_meta)

# Output on icecast
output.icecast(%mp3,mount="test.mp3",s)
```

We can then change the title of the stream by sending OSC messages, for instance

```
oscsend localhost 7777 "/metadata" ss "title" "The new title"
```

CHAPTER 45

Liquidsoap execution phases

There are various stages of running liquidsoap:

- **Parsing**: read scripts and scripting expressions, can fail with syntax errors.
- **Static analysis**: infer the type of all expressions, leaves some type unknown and may fail with type errors.
- **Instantiation**: when script is executed, sources get created. Remaining unknown stream types are forced according to `frame.*.channels` settings, clocks are assigned (but unknown clocks may remain) and some sources are checked to be infallible. Each of these steps may raise an error.
- **Collection**: Unknown clocks become the default wallclock so that all sources are assigned to one clock. Active sources newly attached to clocks are initialized for streaming, shutdown sources are detached from their clocks, and clocks are started or destroyed as needed. Streaming has started.

Usually, liquidsoap is ran by passing one or several scripts and expressions to execute. Those expressions set up some sources, and outputs typically don't change anymore. If those initially provided active sources fail to be initialized (invalid parameter, fail to connect, etc.) liquidsoap will terminate with an error.

It is however possible to **dynamically** create active sources, through registered server commands, event handlers, etc. They will be initialized and run as statically created ones. In **interactive** mode (passing the `--interactive` option) it is also possible to input expressions in a liquidsoap prompt, and their execution can trigger the creation of new outputs.

Outputs can be desactivated using `source.shutdown()`: they will stop streaming and will be destroyed.

The full liquidsoap instance can be shutdown using the `shutdown()` command.

CHAPTER 46

Playlist parsers

Liquidsoap supports various playlists formats. Those formats can be used for `playlist` sources, `input.http` streams and manually using `request.create`.

46.1 Supported formats

Most supported playlists format are *strict*, which means that the decoder can be sure that is has found a correct playlist for that format. Some other format, such as `m3u`, however, may cause *false positive* detections.

All formats are identified by their *mime-type* or *content-type*. Supported formats are the following:

- Text formats:
  - `audio/x-scpls`: PLS format, **strict**
  - `application/x-cue`: CUE format, **strict**. This format's usage is described below.
  - `audio/x-mpegurl`, `audio/mpegurl`: M3U, **non strict**
- Xml formats:
  - `video/x-ms-asf`, `audio/x-ms-asx`: ASX, **strict**
  - `application/smil`, `application/smil+xml`, SMIL, **strict**
  - `application/xspf+xml`, XSPF, **strict**
  - `application/rss+xml`, Podcast, **strict**

46.2 Usage

Playlist files are parsed automatically when used in a `playlist` or `input.http` operator. Each of these two operators has specific options to specify how to pick up a track from the playlist, *e.g.* pick a random track, the first one etc.

Additionally, you can also manually parse and process a playlist using `request.create` and `request.resolve` and some programming magic. You can check the code source for `playlist.reloadable` in our standard library for a detailed example.

46.3 Special case: CUE format

The CUE format originates from CD burning programs. They describe the set of tracks of a whole CD and are accompanied by a single file containing audio data for the whole CD.

This playlist format can be used in liquidsoap, using a `cue_cut` operator. By default, the CUE playlist parser will add metadata from cue-in and cue-out points for each track described in the playlist, which you can then pass to `cue_cut` to play each track of the playlist. Something like:

```
cue_cut(playlist("/path/to/file.cue"))
```

You can find an example of using `cue_cut` with cue sheets here and a throughout explanation of how seeking in liquidsoap works there.

The metadata added for cue-in and cue-out positions can be customized using the following configuration keys:

```
set("playlists.cue_in_metadata","liq_cue_in")
set("playlists.cue_out_metadata","liq_cue_out")
```

CHAPTER 47

header-includes: | \DeclareUnicodeCharacter{03BB}{λ} . . . The theory behind Liquidsoap

47.1 Presentations

47.1.1 ON2 presentation

Savonet was at the ON2: Test Signals conference in Berlin, on October 22-23 2010. We presented Liquidsoap, but also held the first Liquidsoap workshop.

47.2 Publications

47.2.1 Liquidsoap: a High-Level Programming Language for Multimedia Streaming

Many of the advanced features of the Liquidsoap language are described in Liquidsoap: a High-Level Programming Language for Multimedia Streaming. The article details in particular Liquidsoap's handling of heterogeneous stream contents (e.g. audio and video), as well as the model for clocks in the language.

47.2.2 De la webradio lambda à la λ-webradio

The first published presentation of Liquidsoap was made in De la webradio lambda à la λ-webradio (*Baelde D. and Mimram S. in proceedings of Journées Francophnes des Langages Applicatifs (JFLA), pages 47-61, 2008*) – yes, it's in French, sorry. It gives a broad description of the Liquidsoap tool and explains the theory behind the language, which is formalized as a variant of the typed λ-calculus with labels and optional arguments. The article describes the typing inference algorithm as well as some properties of the language (confluence) and of typing (subject reduction, admissible rules, termination of typed terms).

CHAPTER 48

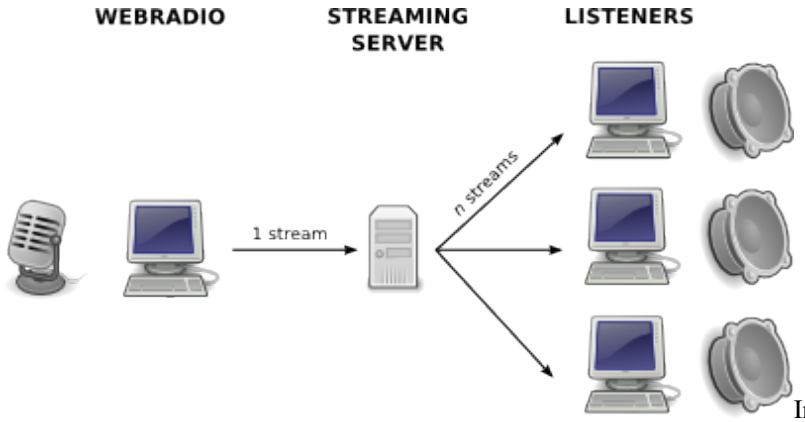
Quickstart

48.1 The Internet radio toolchain

Liquidsoap is a general audio stream generator, but is mainly intended for Internet radios. Before starting with the proper Liquidsoap tutorial let's describe quickly the components of the internet radio toolchain, in case the reader is not familiar with it.

The chain is made of:

- the stream generator (Liquidsoap, ices, or for example a DJ-software running on your local PC) which creates an audio stream (Ogg Vorbis or MP3);
- the streaming media server (Icecast, Shoutcast, ...) which relays several streams from their sources to their listeners;
- the media player (iTunes, Winamp, ...) which gets the audio stream from the streaming media server and plays it to the listener's speakers.



Internet radio toolchainThe stream is always passed from the stream generator to the server, whether or not there are listeners. It is then sent by the server to every listener. The more listeners you have, the more bandwidth you need.

If you use Icecast, you can broadcast more than one audio feed using the same server. Each audio feed or stream is identified by its "mount point" on the server. If you connect to the `foo.ogg` mount point, the URL of your stream will be http://localhost:8000/foo.ogg – assuming that your Icecast is on localhost on port 8000. If you need further information on this you might want to read Icecast's documentation. A proper setup of a streaming server is required for running Liquidsoap.

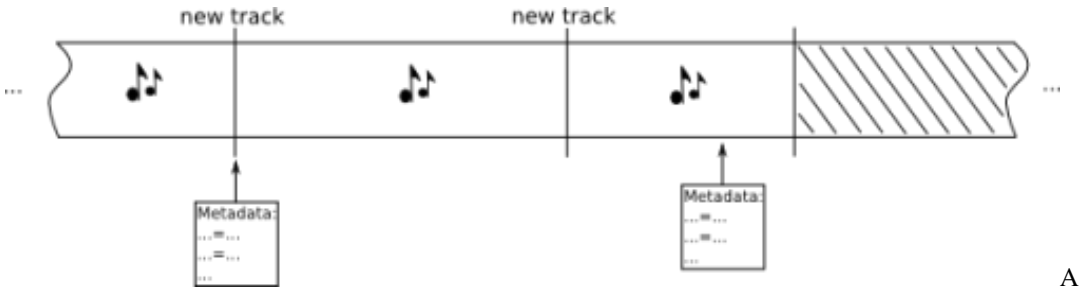
Now, let's create an audio stream.

48.2 Starting to use Liquidsoap

In this tutorial we assume that you have a fully installed Liquidsoap. In particular the library `pervasives.liq` and its accompanying scripts should have been installed, otherwise Liquidsoap won't know the operators which have been defined there. If you installed into the default `/usr/local` you will find it inside `/usr/local/lib/liquidsoap/<version>`.

48.2.1 Sources

A stream is built with Liquidsoap by using or creating sources. A source is an annotated audio stream. In the following picture we represent a stream which has at least three tracks (one of which starts before the snapshot), and a few metadata packets (notice that they do not necessarily coincide with new tracks).



A streamIn a Liquidsoap script, you build source objects. Liquidsoap provides many functions for creating sources from scratch (e.g. `playlist`), and also for creating complex sources by putting together simpler ones (e.g. `switch` in the following example). Some of these functions (typically the `output.*`) create an active source, which will continuously pull its children's stream and output it to speakers, to a file, to a streaming server, etc. These active sources are the roots of a Liquidsoap instance, the sources which bring life into it.

48.2.2 That source is fallible!

A couple of things can go wrong in your streaming system. In Liquidsoap, we say that a source is *infallible* if it is always available. Otherwise, it is *fallible*, something can go wrong. By default, an output requires that its input source is infallible, otherwise it complains that ''That source is fallible!"

For example, a normal `playlist` is fallible. Firstly, because it could contain only invalid files, or at least spend too much time on invalid files for preparing a valid one on time. Moreover, a playlist could contain remote files, which may not be accessible quickly at all times. A queue of user requests is an other example of fallible source. Also, if `file.ogg` is a valid local file, then `single("file.ogg")` is an infallible source.

When an output complains about its source, you have to turn it into an infallible one. Depending on the situation, many solutions are available. The function `mksafe` takes a source and returns an infallible source, streaming silence when the input stream becomes unavailable. In a radio-like stream, silence is not the prefered solution, and you will probably prefer to `fallback` on an infallible ''security" source:

```
fallback([your_fallible_source_here, single("failure.ogg")])
```

Finally, if you do not care about failures, you can pass the parameter `fallible=true` to most outputs. In that case, the output will accept a fallible source, and stop whenever the source fails, to restart when it is ready to emit a stream again. This is usually done if you are not emitting a radio-like stream, but for example capturing or relaying another stream, or encoding files.

48.3 One-line expressions

Liquidsoap is a scripting language. Many simple setups can be achieved by evaluating one-line expressions.

48.3.1 Playlists

In the first example we'll play a playlist. Let's put a list of audio files in `playlist.pls`: one filename per line, lines starting with a # are ignored. You can also put remote files' URLs, if your liquidsoap has support for the corresponding protocols. Then just run:

```
liquidsoap 'out(playlist("playlist.pls"))'
```

Other playlist formats are supported, such as M3U and, depending on your configuration, XSPF. Instead of giving the filename of a playlist, you can also use a directory name, and liquidsoap will recursively look for audio files in it.

Depending on your configuration, the output `out` will use AO, Alsa or OSS, or won't do anything if you do not have support for these libs. In that case, the next example is for you.

48.3.2 Streaming out to a server

Liquidsoap is capable of playing audio on your speakers, but it can also send audio to a streaming server such as Icecast or Shoutcast. One instance of liquidsoap can stream one audio feed in many formats (and even many audio feeds in many formats!).

You may already have an Icecast server. Otherwise you can install and configure your own Icecast server. The configuration typically consists in setting the admin and source passwords, in `/etc/icecast2/icecast.xml`. These passwords should really be changed if your server is visible from the hostile internet, unless you want people to kick your source as admins, or add their own source and steal your bandwidth.

We are now going to send an audio stream, encoded as Ogg Vorbis, to an Icecast server:

```
liquidsoap \
  'output.icecast(%vorbis,
     host = "localhost", port = 8000,
     password = "hackme", mount = "liq.ogg",
     mksafe(playlist("playlist.m3u")))'
```

The main difference with the previous is that we used `output.icecast` instead of `out`. The second difference is the use of the `mksafe` which turns your fallible playlist source into an infallible source.

Streaming to Shoutcast is quite similar, using the `output.shoutcast` function:

```
liquidsoap 'output.shoutcast(%mp3,
                host="localhost", port = 8000,
                password = "changeme",
                mksafe(playlist("playlist.m3u")))'
```

48.3.3 Input from another streaming server

Liquidsoap can use another stream as an audio source. This may be useful if you do some live shows.

```
liquidsoap '
  out(input.http("http://dolebrai.net:8000/dolebrai.ogg"))'
```

48.3.4 Input from the soundcard

If you're lucky and have a working ALSA support, try one of these... but beware that ALSA may not work out of the box.

```
liquidsoap 'output.alsa(input.alsa())'
```

```
liquidsoap 'output.alsa(bufferize = false,
                        input.alsa(bufferize = false))'
```

48.3.5 Other examples

You can play with many more examples. Here are a few more. To build your own, lookup the API documentation to check what functions are available, and what parameters they accept.

```
# Listen to your playlist, but normalize the volume
liquidsoap 'out(normalize(playlist("playlist_file")))'
```

```
# ... same, but also add smart cross-fading
liquidsoap 'out(crossfade(
                  normalize(playlist("playlist_file"))))'
```

48.4 Script files

We have seen how to create a very basic stream using one-line expressions. If you need something a little bit more complicated, they will prove uneasy to manage. In order to make your code more readable, you can write it down to a file, named with the extension `.liq` (eg: `myscript.liq`).

To run the script:

```
liquidsoap myscript.liq
```

On UNIX, you can also put `#!/path/to/your/liquidsoap` as the first line of your script ("shebang"). Don't forget to make the file executable:

```
chmod u+x myscript.liq
```

Then you'll be able to run it like this:

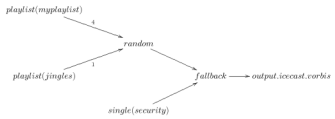
```
./myscript.liq
```

Usually, the path of the liquidsoap executable is `/usr/bin/liquidsoap`, and we'll use this in the following.

48.5 A simple radio

In this section, we build a basic radio station that plays songs randomly chosen from a playlist, adds a few jingles (more or less one every four songs), and output an Ogg Vorbis stream to an Icecast server.

Before reading the code of the corresponding liquidsoap script, it might be useful to visualize the streaming process with the following tree-like diagram. The idea is that the audio streams flows through this diagram, following the arrows. In this case the nodes (`fallback` and `random`) select one of the incoming streams and relay it. The final node `output.icecast` is an output: it actively pulls the data out of the graph and sends it to the world.



Graph for 'basic-radio.liq'

```
#!/usr/bin/liquidsoap
# Log dir
set("log.file.path","/tmp/basic-radio.log")

# Music
myplaylist = playlist("~/radio/music.m3u")
# Some jingles
jingles = playlist("~/radio/jingles.m3u")
# If something goes wrong, we'll play this
security = single("~/radio/sounds/default.ogg")

# Start building the feed with music
radio = myplaylist
# Now add some jingles
radio = random(weights = [1, 4],[jingles, radio])
# And finally the security
radio = fallback(track_sensitive = false, [radio, security])

# Stream it out
output.icecast(%vorbis,
  host = "localhost", port = 8000,
  password = "hackme", mount = "basic-radio.ogg",
  radio)
```

48.6 What's next?

You can first have a look at a more complex example. There is also a second tutorial about advanced techniques.

You should definitely learn how to get help. If you know enough liquidsoap for your use, you'll only need to refer to the scripting reference, or see the cookbook. At some point, you might read more about Liquidsoap's scripting language. For a better understanding of liquidsoap, it is also useful to read a bit about the notions of sources and requests.

CHAPTER 49

An advanced script to listen to radio nova.

49.1 Introduction

Radio Nova is a french eclectic radio that provides a mp3 stream via Icecast. However, they don't include the metadata (probably to avoid rippers..).

Still, these metadata are available on their webpage:

```
14:29 toots@leonard /tmp% wget -q http://www.novaplanet.com -O - | grep 'scroll_play'
   scroll_play("ESTHER PHILLIPS","DISPOSABLE SOCIETY", 'playing_now', '0');
```

Hence, it would be nice to dynamically grab these metadata and insert it in the stream. The following script will perform this.

49.2 The script

```
#!/usr/bin/liquidsoap
# Liquidsoap script to listen to radio nova, grabbing metadata
# on the webpage.

# Disable file log, enable stdout log
set("log.file",false)
set("log.stdout",true)

# Initial input.http source
nova = input.http("http://broadcast.infomaniak.net:80/radionova-high.mp3")
# Remove metadata, add a hook to insert new metadata
nova = insert_metadata(id="nova",
         clear_metadata(nova))

# This string references will be used to keed track
# of previous metadata
```

(continues on next page)

(continued from previous page)

```
title = ref "unknown title"
artist = ref "unknown artist"

# Capitalize and lowercase
def cap(s)
  string.capitalize(string.case(s))
end

# Process to grab metadata on the webpage
# Returns "artist","title"
def metas() =
  s = list.hd(
        default="",
        get_process_lines(
           "wget -q http://www.novaplanet.com -O - | grep 'scroll_play'"))
  s = string.extract(pattern='scroll_play\("([^"]*)",\s*"([^"]*)"',s)
  (cap(list.assoc(default="","1",s)),cap(list.assoc(default="","2",s)))
end

# Process that inserts grabbed metadata
# to the stream
def add_meta_nova()
  log = log(level=4)
  log("Checking for metas")
  x = metas()
  new_artist = fst(x)
  new_title = snd(x)
  old_title = !title
  old_artist = !artist
  if (old_artist != new_artist or old_title != new_title) and
     (new_title != "" or new_artist != "")
  then
    log("Got new metas: #{new_artist} -- #{new_title}")
    ignore(
      server.execute(
        'nova.insert artist="#{new_artist}",title="#{new_title}"'))
    title := new_title
    artist := new_artist
  else
    log("Keeping old metas")
  end
  0.2
end

# Schedule the insert process every 0.2 second
add_timeout(fast=false,0.2,add_meta_nova)

# function to display new metadatas
def print_meta(m) =
  def print(z) =
    label = fst(z)
    value = snd(z)
    log("Metadata: #{label}=#{value}")
  end
  list.iter(print,m)
end
```

(continues on next page)

(continued from previous page)

```
# Hook the previous function on the stream
nova = on_metadata(print_meta,nova)

# Play the stream !
out(nova)
```

CHAPTER 50

RadioPi

RadioPi is the web radio of the ECP (Ecole Centrale de Paris). RadioPi runs many channels. There are topical channels (Reggae, Hip-Hop, Jazz, ...). On top of that, they periodically broadcast live shows, which are relayed on all channels.

We met a RadioPi manager right after having released Liquidsoap 0.2.0, and he was seduced by the system. They needed quite complex features, which they were at that time fulfilling using dirty tricks, loads of obfuscated scripts. Using Liquidsoap now allow them to do all they want in an integrated way, but also provided new features.

50.1 The migration process

Quite easy actually. They used to have many instances Ices2, each of these calling a Perl script to get the next song. Other scripts were used for switching channels to live shows.

Now they have this single Liquidsoap script, no more. It calls external scripts to interact with their web-based song scheduling system. And they won new features: blank detection and distributed encoding.

The first machine gets its files from a ftp server opened on the second machine. Liquidsoap handles download automatically.

Each file is given by an external script, `radiopilote-getnext`, whose answer looks as follows (except that it's on a single line):

```
annotate:file_id="3541",length="400.613877551",\
  type="chansons",title="John Holt - Holigan",\
  artist="RadioPi - Canal reggae",\
  album="Studio One SeleKta! - Album Studio 1 12",\
  canal="reggae":ftp://***:***@host/files/3541.mp3
```

Note that we use annotate to pass some variables to liquidsoap...

```
#!/usr/bin/liquidsoap

# Standard settings
set("log.file.path","/var/log/liquidsoap/pi.log")
```

(continues on next page)

(continued from previous page)

```
set("init.daemon",true)
set("log.stdout",false)
set("log.file",true)
set("init.daemon.pidfile.path","/var/run/liquidsoap/pi.pid")

# Enable telnet server
set("server.telnet",true)

# Enable harbor for any external
# connection
set("harbor.bind_addr","0.0.0.0")

# Verbose logs
set("log.level",4)

# We use the scheduler intensively,
# therefore we create many queues.
set("scheduler.generic_queues",5)
set("scheduler.fast_queues",3)
set("scheduler.non_blocking_queues",3)

# === Settings ===

# The host to request files
stream = "XXXxXXXx"
# The command to request files
scripts = "ssh XXxxxXXX@#{stream} '/path/to/scripts/"
# A substitution on the returned path
sed = " | sed -e s#/path/to/files/#ftp://user:password@#{stream}/#'"

# Enable replay gain
enable_replaygain_metadata ()

pass = "XXxXXXXx"
ice_host = "localhost"

descr = "RadioPi"
url = "http://radiopi.org"

# === Live ===

# A live source, on which we stip blank (make the source
# unavailable when streaming blank)
live =
  strip_blank(
    input.harbor(id="live", port=8000, password=pass,
                 buffer=8.,max=20.,"live.ogg"),
    length=10., threshold=-50.)

# This source relays the live data, when available,
# to the other streamer, in uncompressed format (WAV)
output.icecast(%wav, host=stream,
               port=8005, password=pass,
               mount="live.ogg", fallible=true,
               live)

# This source relays the live source to "live.ogg". This
```

(continues on next page)

(continued from previous page)

```
# is used for debugging purposes, to see what is sent
# to the harbor source.
output.icecast(%vorbis, host="127.0.0.1",
               port=8080, password=pass,
               mount="live.ogg", fallible=true,
               live)

# This source starts an archive of the live stream
# when available
title = '$(if $(title),"$(title)",\
         "Emission inconnue")$(if $(artist), \
         " par  $(artist)") - %m-%d-%Y, %H:%M:%S'
output.file(%vorbis, reopen_on_metadata=true,
            fallible=true,
            "/data/archives/brutes/" ^ title ^ ".ogg",
            live)

# === Channels ===

# Specialize the output functions by partial application
output.icecast = output.icecast(description=descr, url=url)
out = output.icecast(host=ice_host,port=8080,password=pass,fallible=true)
out_aac32 = out(%fdkaac(bitrate=32))
out_aac = out(%fdkaac(bitrate=64))
out = out(%mp3)

# A file for playing during failures
interlude =
  single("/home/radiopi/fallback.mp3")

# Lastfm submission
def lastfm (m) =
  if (m["type"] == "chansons") then
    if (m["canal"] == "reggae" or m["canal"] == "Jazz" or m["canal"] == "That70Sound
↪") then
      canal =
        if (m["canal"] == "That70Sound") then
           "70sound"
        else
           m["canal"]
        end
      user = "radiopi-" ^ canal
      lastfm.submit(user=user,password="xXXxx",m)
    end
  end
end

# === Basic sources ===

# Custom crossfade to deal with jingles..
def crossfade (~start_next=5.,~fade_in=3.,~fade_out=3.,
               ~default=(fun (a,b) -> sequence([a, b])),
               ~high=-15., ~medium=-32., ~margin=4.,
               ~width=2.,~conservative=false,s)
  fade.out = fade.out(type="sin",duration=fade_out)
  fade.in  = fade.in(type="sin",duration=fade_in)
  add = fun (a,b) -> add(normalize=false,[b, a])
```

(continues on next page)

(continued from previous page)

```
log = log(label="crossfade")

def transition(a,b,ma,mb,sa,sb)

  list.iter(fun(x)-> log(level=4,"Before: #{x}"),ma)
  list.iter(fun(x)-> log(level=4,"After : #{x}"),mb)

  if ma["type"] == "jingles" or mb["type"] == "jingles" then
    log("Old or new file is a jingle: sequenced transition.")
    sequence([sa, sb])
  elsif
    # If A and B are not too loud and close, fully cross-fade them.
    a <= medium and b <= medium and abs(a - b) <= margin
  then
    log("Old <= medium, new <= medium and |old-new| <= margin.")
    log("Old and new source are not too loud and close.")
    log("Transition: crossed, fade-in, fade-out.")
    add(fade.out(sa),fade.in(sb))

  elsif
    # If B is significantly louder than A, only fade-out A.
    # We don't want to fade almost silent things, ask for >medium.
    b >= a + margin and a >= medium and b <= high
  then
    log("new >= old + margin, old >= medium and new <= high.")
    log("New source is significantly louder than old one.")
    log("Transition: crossed, fade-out.")
    add(fade.out(sa),sb)

  elsif
    # Opposite as the previous one.
    a >= b + margin and b >= medium and a <= high
  then
    log("old >= new + margin, new >= medium and old <= high")
    log("Old source is significantly louder than new one.")
    log("Transition: crossed, fade-in.")
    add(sa,fade.in(sb))

  elsif
    # Do not fade if it's already very low.
    b >= a + margin and a <= medium and b <= high
  then
    log("new >= old + margin, old <= medium and new <= high.")
    log("Do not fade if it's already very low.")
    log("Transition: crossed, no fade.")
    add(sa,sb)

  # What to do with a loud end and a quiet beginning ?
  # A good idea is to use a jingle to separate the two tracks,
  # but that's another story.

  else
    # Otherwise, A and B are just too loud to overlap nicely,
    # or the difference between them is too large and overlapping would
    # completely mask one of them.
    log("No transition: using default.")
    default(sa, sb)
```

(continues on next page)

(continued from previous page)

```
    end
  end

  cross(width=width, duration=start_next, conservative=conservative,
              transition,s)
end

# Create a radiopilote-driven source
def channel_radiopilote(~skip=true,name)
  log("Creating canal #{name}")

  # Request function
  def request () =
    log("Request for #{name}")
    ret = list.hd(get_process_lines(scripts^"radiopilote-getnext "^quote(name)^sed))
    log("Got answer: #{ret} for #{name}")
    request.create(ret)
  end

  # Create the request.dynamic source
  # Set conservative to true to queue
  # several songs in advance
  source =
    request.dynamic(conservative=true, length=400.,
                    id="dyn_"^name,request,
                    timeout=60.)

  # Apply normalization using replaygain
  # information
  source =   amplify(1.,override="replay_gain", source)

  # Skip blank when asked to
  source =
    if skip then
      skip_blank(source, length=10., threshold=-40.)
    else
      source
    end

  # Submit new tracks on lastfm
  source = on_metadata(lastfm,source)

  # Tell the system when a new track
  # is played
  source = on_metadata(fun (meta) ->
                    system(scripts ^ "radiopilote-feedback "
                           ^quote(meta["canal"])^" "
                           ^quote(meta["file_id"]) ^ "'"), source)

  # Finally apply a smart crossfading
  crossfade(source)
end

# Basic source
jazz = channel_radiopilote("jazz")
discoqueen = channel_radiopilote("discoqueen")
# Avoid skiping blank with classic music !!
```

(continues on next page)

(continued from previous page)

```
classique = channel_radiopilote(skip=false,"classique")
That70Sound = channel_radiopilote("That70Sound")
metal = channel_radiopilote("metal")
reggae = channel_radiopilote("reggae")
Rock = channel_radiopilote("Rock")

# Group those sources in a seperate
# clock (good for multithreading/multicore)
clock.assign_new([jazz,That70Sound,metal,reggae])

# === Mixing live ===

# To create a channel from a basic source, add:
# - a new-track notification for radiopilote
# - metadata rewriting
# - the live shows
# - the failsafe 'interlude' source to channels
# - blank detection
def mklive(source) =
  # Transition function: if transitioning
  # to the live, fade out the old source
  # if transitioning from live, fade.in
  # the new source. NOTE: We cannot
  # skip the current song because
  # reloading new songs for all the
  # sources when live starts costs too much
  # CPU.
  def trans(old,new) =
    if source.id(new) == source.id(live) then
      log("Transition to live!")
      add([new,fade.out(old)])
    elsif source.id(old) == source.id(live) then
      log("Transitioning from live!")
      add([fade.in(new),old])
    else
      log("Dummy transition")
      new
    end
  end
  fallback(track_sensitive=false,
    transitions=[trans,trans,trans],
    [live,source,interlude])
end

# Create a channel using mklive(), encode and output it to icecast.
def mkoutput(~out=out,mount,source,name,genre)
  out(id=mount,mount=mount,name=name,genre=genre,
     mklive(source)
     )
end

# === Outputs ===

mkoutput("jazz", jazz, "RadioPi - Canal Jazz","jazz")
mkoutput("discoqueen", discoqueen, "RadioPi - Canal DiscoQueen","discoqueen")
mkoutput("classique", classique, "RadioPi - Canal Classique","classique")
mkoutput("That70Sound", That70Sound,
```

(continues on next page)

(continued from previous page)

```
                        "RadioPi - Canal That70Sound","That70Sound")
mkoutput("metal", metal, "RadioPi - Canal Metal","metal")
mkoutput("reggae", reggae, "RadioPi - Canal Reggae","reggae")
mkoutput("Rock", Rock, "RadioPi - Canal Rock","Rock")

# Test outouts
mkoutput(out=out_aac,"reggae.aacp", reggae, "RadioPi - Canal Reggae \
                                      (64 kbits AAC+ test stream)","reggae")
mkoutput(out=out_aac32,"reggae.aacp32", reggae, "RadioPi - Canal Reggae \
                                       (32 kbits AAC+ test stream)","reggae")
```

The other machine has a similar configuration exept that files are local, but this is exactly the same for liquidsoap !

Using harbor, the live connects directly to liquidsoap, using port `8000` (icecast runs on port `8080`). Then, liquidsoap starts a relay to the other encoder, and both switch their channels to the new live.

Additionally, a file output is started upon live connection, in order to backup the stream. You could also add a relay to icecast in order to manually check what's received by the harbor.

CHAPTER 51

Normalization and replay gain

51.1 Normalization

If you want to have a constant average volume on an audio stream, you can use the `normalize` operator. However, this operator cannot guess the volume of the whole stream, and can be ‘‘surprised” by rapide changes of the volume. This can lead to a volume that is too low, too high, oscillates. In some cases, dynamic normalization also creates saturation.

To tweak the normalization, several parameters are available. These are listed and explained in the reference and also visible by executing `liquidsoap -h normalize`. However, if the stream you want to normalize consist of audio files, using the replay gain technology might be a better choice.

51.2 Replay gain

Replay gain is a proposed standard that is (more or less) respected by many open-source tools. It provides a way to obtain an overall uniform perceived loudness over a track or a set of tracks. The computation of the loudness is based on how the human ear actually perceives each range of frequency. Having computed the average perceived loudness on a track or an album, it is easy to renormalize the tracks when playing, ensuring a comfortable, consistent listening experience.

Because it is track-based, replay gain does not suffer from the typical problems of stream-based, dynamic approaches. Namely, these distort the initial audio, since they constantly adapt the amplification factor. Sometimes it oscillates too quickly in a weird audible way. Sometimes it does not adapt quickly enough, leading to under or over-amplified sections.

On the other hand, replay gain has its drawbacks. First, it requires an initial computation that is a bit costly. This computation can be done once for all for local files – subsequent calls can then retrieve the result from the metadata. Although not impossible in theory, there is no recipe for liquidsoap to offer the same feature on remote files.

51.2.1 How to use replay gain in Liquidsoap

One easy way to enable replay gain is to use the `ffmpeg` protocol:

```
set("protocol.ffmpeg.replaygain",true)

s = single("ffmpeg:/path/to/file.mp3")

...
```

With the `ffmpeg` protocol, files are entirely decoded to `WAV` format using the `ffmpeg` binary. When the `"protocol.ffmpeg.replaygain"` setting is set to `true`, it will also apply the replay gain amplification while decoding.

The protocol requires `ffmpeg` in the path, which can be set via

```
set("protocol.ffmpeg.path","...")
```

The `ffmpeg` protocol is handy but it is also limited. For instance, it decodes and analyzes whole files, which can be a problem if you are using large audio files. Aditionally, you might want to exclude certain files, e.g. jingles from this processing.

If you need more finer-grained control or do not wish to use the `ffmpeg` protocol, you can see the method below.

51.2.2 Renormalizing according to some metadata field

The `amplify()` operator can behave according to metadata. Its `override` parameter indicates a metadata field that, when present and well-formed, overrides the amplification factor. Well formed fields are floats (e.g. `2` or `0.7`) for linear amplification factors and floats postfixed with `dB` (e.g. `-2 dB`) for logarithmic ones.

For replay gain implementation, the `amplify` operator would typically be added immediately on top of the basic tracks source, before transitions or other audio processing operators. We follow these lines in the next example, where the `replay_gain` field is used to carry the information:

```
list    = playlist("~/playlist")
default = single("~/default.ogg")

s = fallback([list,default])
s = amplify(1.,override="replay_gain",s)

# Here: other effects, and finally the output...
```

You may also take care of not losing the information brought by the metadata. This may be the case for instance if you use `crossfade` before applying normalization. Hence, normalization should be done as soon as possible in the script, if possible just after the initial source.

51.2.3 Computing and retrieving the data

In practice, the replay gain information can be found in various fields depending on the audio format and the replay gain computation tool.

Liquidsoap provides a script for extracting the replay gain value which requires `ffmpeg`.

There are at least two ways to use it in your liquidsoap script:

using the replay gain metadata resolver, or the `replay_gain` protocol.

The metadata solution is uniform: without changing anything, *all* your files will have a new `replay_gain` metadata when the computation suceeded. However, this can be problematic, for example, for jingles, or if you have large files that would take a very long time to be analyzed by replaygain tools. The protocol solution gives you more control on when the replaygain analysis is performed, but requires that you change some `uri` into `replay_gain:uri`. We briefly discuss below how to do it conveniently in some typical cases.

Note that our replaygain support for remote files can be problematic. As such, it would analyze the file after each download, which may be uselessly costly. One should instead make sure that the file has been analyzed on the remote machine, so that the local analysis only retrieves the precomputed value. In any case, remote files can only be treated through the addition of a metadata resolver, and cannot work with the `replay_gain` protocol technique (`replaygain:ftp://host/file.ogg` will call the script using the `ftp://host/file.ogg` as the URI parameter, and it will fail).

The replay gain metadata resolver is not enabled by default. You can do it by adding the following code in your script:

```
enable_replaygain_metadata()
```

The `replay_gain` protocol is enabled by default. In this case, everytime you need replaygain information about a file, access it through your new protocol: for example, replace `/path/to/file.mp3` by `replay_gain:/path/to/file.mp3`. The resolving of the protocol will trigger a call to our script, which will return an annotated request, finally resulting in your file with the extra `replay_gain` metadata.

Prepending `replay_gain:` is easy if you are using a script behind some `request.dynamic` operator. If you are using the `playlist` operator, you can use its `prefix` parameter.

CHAPTER 52

An abstract notion of files: requests

The request is an abstract notion of file which can be conveniently used for defining powerful sources. A request can denote a local file, a remote file, or even a dynamically generated file. They are resolved to a local file thanks to a set of *protocols*. Then, audio requests are transparently decoded thanks to a set of audio and metadata *formats*.

The systematic use of requests to access files allows you to use remote URIs instead of local paths everywhere. It is perfectly OK to create a playlist for a remote list containing remote URIs: `playlist("http://my/friends/playlist.pls")`.

52.1 The resolution process

The nice thing about resolution is that it is recursive and supports backtracking. An URI can be changed into a list of new ones, which are in turn resolved. The process succeeds if some valid local file appears at some point. If it doesn't succeed on one branch then it goes back to another branch. A typical complex resolution would be:

- `bubble:artist="bodom"` * `ftp://no/where` * `Error`
- `ftp://some/valid.ogg` * `/tmp/success.ogg`

On top of that, metadata is extracted at every step in the branch. Usually, only the final local file yields interesting metadata (artist,album,...). But metadata can also be the nickname of the user who requested the song, set using the `annotate` protocol.

At the end of the resolution process, in case of a media request, liquidsoap checks that the file is decodable, *i.e.*, there should be a valid decoder for it.

Each request gets assigned a request identifier (RID) which is used by various sources to identify which request(s) they are using. Knowing this number, you can monitor a request, even after it's been destroyed (see setting `request.grace_time`). Two server commands are available: `request.trace` shows a log of the resolution process and `request.metadata` shows the current request metadata. In addition, server commands are available to obtain the list of all requests, alive requests, currently resolving requests and currently playing requests (respectively `request.all`, `request.alive`, `request.resolving`, `request.on_air`).

52.2 Currently supported protocols

- HTTP, HTTPS, FTP thanks to curl
- SAY for speech synthesis (requires festival): `say:I am a robot` resolves to the WAV file resulting from the synthesis.
- TIME for speech synthesis of the current time: `time: It is exactly $(time), and you're still listening.`
- ANNOTATE for manually setting metadata, typically used in `annotate:nick="alice", message="for bob":/some/track/uri`

The extra metadata can then be synthesized in the audio stream, or merged into the standard metadata fields, or used on a rich web interface... It is also possible to add a new protocol from the script, as it is done with bubble for getting songs from a database query.

52.3 Currently supported formats

- MPEG-1 Layer II (MP2) and Layer III (MP3) through libmad and `ocaml-mad`
- Ogg Vorbis through libvorbis and `ocaml-vorbis`
- WAV
- AAC
- and much more through external decoders!

Playing files is the most common way to build an audio stream. In liquidsoap, files are accessed through requests, which combine the retrieval of a possibly remote file, and its decoding.

Liquidsoap provides several operators for playing requests: `single`, `playlist` and `playlist.safe`, `request.dynamic`, `request.queue` and `request.equeue`. In a few cases (`single` with a local file, or `playlist.safe`) a request operator will know that it can always get a ready request instantaneously. It will then be infallible. Otherwise, it will have a queue of requests ready to be played (local files with a valid content), and will feed this queue in the background. This process is described here.

CHAPTER 53

Common parameters

Queued request sources maintain an *estimated remaining time*, and trigger a new request resolution when this remaining time goes below their `length` parameter.

The estimation is based on the duration of files prepared in the queue, and the estimated remaining time in the currently playing file. Precise file durations being expensive to compute, they are not forced: if a duration is provided in the metadata it shall be used, otherwise the `default_length` is assumed.

For example, with the default 10 seconds of wanted queue length, the operator will only prepare a new file 10 seconds before the end of the current one.

Up to liquidsoap 0.9.1, the estimated remaining time in the current track was not taken into account. With this behavior, each request-based source would keep at least one song in queue, which was sometimes inconvenient. This behavior can be restored by passing `conservative=true`, which is useful in some cases: it helps to ensure that a song will be ready in case of skip; generally, it prepares things more in advance, which is good when resolution is long (*e.g.*, heavily loaded server, remote files).

CHAPTER 54

Request.dynamic

This source takes a custom function for creating its new requests. This function, of type `()->request`, can for example call an external program.

To create the request, the function will have to use the `request.create` function which has type `(string,?indicators:[string])`. The first string is the initial URI of the request, which is resolved to get an audio file. The second argument can be used to directly specify the first row of URIs (see the page about requests for more details), in which case the initial URI is just here for naming, and the resolving process will try your list of indicators one by one until a valid audio file is obtained.

An example that takes the output of an external script as an URI to create a new request can be:

```
def my_request_function() =
  # Get the first line of my external process
  result =
    list.hd(
      get_process_lines("my_script my_params"))
  # Create and return a request using this result
  request.create(result)
end

# Create the source
s = request.dynamic(my_request_function)
```

CHAPTER 55

Queues

Liquidsoap features two sources which provide request queues that can be directly manipulated by the user, via the server interface: `request.queue` and `request.equeue`. The former is a queued source where you can only push new requests, while the later can be edited.

Both operators actually deal with two queues: *primary* and *secondary* queues. The secondary queue is user-controlled. The primary queue is the one that all queued request sources have, its behavior is the same as described above, and it cannot be changed in any way by the user. Requests added to the secondary queue sit there until the feeding process gets them and attempts to prepare them and put them in the primary queue. You can set how many requests will be in that primary queue by tweaking the common parameters of all queued request sources.

The two sources are controlled via the command server. They both feature commands for looking up the queues, queuing new requests, and the `equeue` operator also allows removal and exchange of requests in the secondary queue.

CHAPTER 56

Script loading

When you run liquidsoap for streaming, the command line has the following form:

```
$ liquidsoap script_or_expr_1 ... script_or_expr_N
```

This allows you to ask liquidsoap to load definition and settings from some scripts so that the become available when processing the next ones.

For example you can store your passwords by defined the variable `xxx` in `secret.liq`, and then refer to that variable in your main script `main.liq`. You would then run `liquidsoap secret.liq main.liq`. If you ever need to communicate `main.liq` there won't be any risk of divulgating your password.

56.1 The pervasive script library

In fact, liquidsoap also implicitly loads scripts before those that you specify on the command-line. These scripts are meant to contain standard utilities. Liquidsoap finds them in `LIBDIR/liquidsoap/VERSION` where `LIBDIR` depends on your configuration (it is typically `/usr/local/lib` or `/usr/lib`) and `VERSION` is the version of liquidsoap (*e.g.* `0.3.8` or `svn`).

Currently, liquidsoap loads `pervasives.liq` from the library directory, and this file includes some others. You can add your personal standard library in that directory if you find it useful.

CHAPTER 57

Seeking in liquidsoap

Starting with Liquidsoap `1.0.0-beta2`, it is now possible to seek within sources! Not all sources support seeking though: currently, they are mostly file-based sources such as `request.queue`, `playlist`, `request.dynamic` etc..

The basic function to seek within a source is `source.seek`. It has the following type:

```
(source('a),float)->float
```

The parameters are:

- The source to seek.
- The duration in seconds to seek from current position.

The function returns the duration actually seeked.

Please note that seeking is done to a position relative to the *current* position. For instance, `source.seek(s,3.)` will seek 3 seconds forward in source `s` and `source.seek(s,(-4.))` will seek 4 seconds backward.

Since seeking is currently only supported by request-based sources, it is recommended to hook the function as close as possible to the original source. Here is an example that implements a server/telnet seek function:

```
# A playlist source
s = playlist("/path/to/music")

# The server seeking function
def seek(t) =
  t = float_of_string(default=0.,t)
  log("Seeking #{t} sec")
  ret = source.seek(s,t)
  "Seeked #{ret} seconds."
end

# Register the function
server.register(namespace=source.id(s),
                description="Seek to a relative position \
```

(continues on next page)

(continued from previous page)

```
                              in source #{source.id(s)}",
                usage="seek <duration>",
                "seek",seek)
```

57.1 Cue points

Sources that support seeking can also be used to implement cue points. The basic operator for this is `cue_cut`. Its has type:

```
(?id:string,?cue_in_metadata:string,
 ?cue_out_metadata:string,
 source(audio='#a,video='#b,midi='#c))->
    source(audio='#a,video='#b,midi='#c)
```

Its parameters are:

- `cue_in_metadata`: Metadata for cue in points, default: `"liq_cue_in"`.
- `cue_out_metadata`: Metadata for cue out points, default: `"liq_cue_out"`.
- The source to apply cue points to.

The values of cue-in and cue-out points are given in absolute position through the source's metadata. For instance, the following source will cue-in at 10 seconds and cue-out at 45 seconds on all its tracks:

```
s = playlist(prefix="annotate:liq_cue_in=\"10.\",liq_cue_out=\"45\":",
             "/path/to/music")

s = cue_cut(s)
```

As in the above example, you may use the `annotate` protocol to pass custom cue points along with the files passed to Liquidsoap. This is particularly useful in combination with `request.dymanic` as an external script can build-up the appropriate URI, including cue-points, based on information from your own scheduling back-end.

Alternatively, you may use `map_metadata` to add those metadata. The operator `map_metadata` supports seeking and passes it to its underlying source.

CHAPTER 58

Interaction with the server

Liquidsoap starts with one or several scripts as its configuration, and then streams forever if everything goes well. Once started, you can still interact with it by means of the *server*. The server allows you to run commands. Some are general and always available, some belong to a specific operator. For example the `request.queue()` instances register commands to enqueue new requests, the outputs register commands to start or stop the outputting, display the last ten metadata chunks, etc.

The protocol of the server is a simple human-readable one. Currently it does not have any kind of authentication and permissions. It is currently available via two media: TCP and Unix sockets. The TCP socket provides a simple telnet-like interface, available only on the local host by default. The Unix socket interface (*cf.* the `server.socket` setting) is through some sort of virtual file. This is more constraining, which allows one to restrict the use of the socket to some priviledged users.

You can find more details on how to configure the server in the documentation of the settings key `server`, in particular `server.telnet` for the TCP interface and `server.socket` for the Unix interface. Liquidsoap also embeds some documentation about the available server commands.

Now, we shall simply enable the Telnet interface to the server, by setting `set("server.telnet",true)` or simply passing the `-t` option on the command-line. In a complete case analysis we set up a `request.queue()` instance to play user requests. It had the identifier `"queue"`. We are now going to interact via the server to push requests into that queue:

```
dbaelde@selassie:~$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
queue.push /path/to/some/file.ogg
5
END
request.metadata 5
[...]
END
queue.push http://remote/audio.ogg
6
END
```

(continues on next page)

(continued from previous page)

```
request.trace 6
[...see if the download started/succeeded...]
END
exit
```

Of course, the server isn't very user-friendly. But it is easy to write scripts to interact with Liquidsoap in that way, to implement a website or an IRC interface to your radio. However, this sort of tool is often bound to a specific usage, so we have not released any of ours. Feel free to ask the community about code that you could re-use.

58.1 Interactive variables

Sometimes it is useful to control a variable using telnet. A simple way to achive this is to use the `interactive.float` function. For instance, in order to dynamically the volume of a source:

```
# Register a telnet variable named volume with 1 as initial value
v = interactive.float("volume", 1.)

# Change the volume accordingly
source = amplify(v, source)
```

The first line registers the variable volume on the telnet. Its value can be changed using the telnet command

```
var.set volume = 0.5
```

and it can be retrieved using

```
var.get volume
```

Similarly, we can switch between two tracks using `interactive.bool` and `switch` as follows:

```
# Activate the telnet server
set("server.telnet",true)

# The two sources
s1 = playlist("~/Music")
s2 = sine()

# Create an interactive boolean
b = interactive.bool("button", true)

# Switch between the tracks depending on the boolean
s = switch(track_sensitive=false,[(b,s1), ({true},s2)])

# Output the result
output.pulseaudio(s)
```

By default the source s1 is played. To switch to s2, you can connect on the telnet server and type `var.set button = false`.

58.2 Interactive commands

Starting with liquidsoap version `1.3.4`, you can register custom server commands to interact with the client with applications such as implementing a `pub/sub` mechanism.

There main commands are:

- `server.write`, `server.read`, `server.readchars` and `server.readline` to read and write interactively
- `server.condition`, `server.wait`, `server.signal` and `server.broadcast` to control the execution of the command

58.2.1 Read/Write

Writing a partial response is done using the following syntactic sugar:

```
server.write "string to write" then
  log("string done writting!")
  # Do more stuff then send the final response:
  "Done!"
end
```

Read a value can be done 3 different ways. Most simple one is `server.readline`:

```
server.readline ret then
  log("Read line: #{ret}")
  # Do more stuff then send the final response:
  "Done!"
end
```

Then you can read a fixed number of characters:

```
server.readchars 15 : ret then
  log("Read 15 characters: #{ret}")
  # Do more stuff then send the final response:
  "Done!"
end
```

Finally, you can read until reaching a marker, which can be any string or regular expression:

```
server.read "OVER[\r\n]+" :  ret then
  log("Read until OVER: #{ret}")
  # Do more stuff then send the final response:
  "Done!"
end
```

58.2.2 Control flow

You can pause and resume server commands using an API similar to Unix conditions:

- `server.condition()` creates a condition variable
- `server.wait` pauses a server command. See below for details
- `server.signal(c)` resumes one waiting command
- `server.broadcast(c)` resumes all waiting commands

`server.wait` is used through a syntactic sugar:

```
server.wait c then
  log("Command has resumed!")
  # Do more stuff then send the final response:
  "Done!"
end
```

58.2.3 Full example

In the following, we define two commands:

- `wait`: when executing the command, the client waits for a message. Message can be one of:
  - `"exit"`: terminate command
  - `"read"`: read one line from the client and print it back
  - Otherwise, the client prints the received value
- `send <value>`: when executing this command, the client sends `<value>` to all waiting clients.

```
c = server.condition()

value = ref ""

def wait(_) =
  def rec fn () =
    server.write ">> " then
      server.wait c then
        value = !value
        if value == "exit" then
          "All done!"
        elsif value == "read" then
          server.write "Write me sumething mister..\n" then
            server.readline ret then
              server.write "Read: #{ret}\n" then
                fn()
              end
            end
          end
        else
          server.write "Received value: #{value}\n" then
            fn()
          end
        end
      end
    end
  end

  fn ()
end

def send(v) =
  value := v
  server.signal(c)
  "Ok!"
end
```

Example of use:

send:

```
Connected to localhost.
Escape character is '^]'.
send foo
Ok!
END
send read
Ok!
END
send exit
Ok!
END
```

wait:

```
Connected to localhost.
Escape character is '^]'.
wait
>> Received value: foo
>> Write me sumething mister..
Here's to you mon ami!
Read: Here's to you mon ami!
>> All done!
END
exit
Bye!
```

58.3 Securing the server

The command server provided by liquidsoap is very convenient for manipulating a running instance of Liquidsoap. However, no authentication mechanism is provided. The telnet server has no authentication and listens by default on the localhost (`127.0.0.1`) network interface, which means that it is accessible to any logged user on the machine.

Many users have expressed interest into setting up a secured access to the command server, using for instance user and password information. While we understand and share this need, we do not believe this is a task that lies into Liquidsoap's scope. An authentication mechanism is not something that should be implemented naively. Being SSH, HTTP login or any other mechanism, all these methods have been, at some point, exposed to security issues. Thus, implementing our own secure access would require a constant care about possible security issues.

Rather than doing our own home-made secure acces, we believe that our users should be able to define their own secure access to the command server, taking advantage of a mainstream authentication mechanism, for instance HTTP or SSH login. In order to give an example of this approach, we show here how to create a SSH access to the command server: we create a SSH user that, when logging through SSH, has only access to the command server.

First, we enable the unix socket for the command server in Liquidsoap:

```
set("server.socket",true)
set("server.socket.path","/path/to/socket")
```

When started, liquidsoap will create a socket file `/path/to/socket` that can be used to interact with the command server. For instance, if your user has read and write rights on the socket file, you can do

```
socat /path/to/socket -
```

The interface is then exactly the same has for the telnet server.

We define now a new ‘‘shell”. This shell is in fact the invokation of the socat command. Thus, we create a `/usr/local/bin/liq_shell` file with the following content:

```
#!/bin/sh
# We test if the file is a socket, readable and writable.
if [ -S /path/to/socket ] && [ -w /path/to/socket ] && \
   [ -r /path/to/socket ]; then
  socat /path/to/socket -
else
# If not, we exit..
  exit 1
fi
```

We set this file as executable, and we add it in the list of shells in `/etc/shells`.

Now, we create a user with the `liq_shell` as its shell:

```
adduser --shell /usr/local/bin/liq_shell liq-user
```

You also need to make sure that `liq-user` has read and write rights on the socket file.

Finally, when logging through ssh with `liq-user`, we get:

```
11:27 toots@leonard % ssh liq-user@localhost
liq-user@localhost's password:
Linux leonard 2.6.32-4-amd64 #1 SMP Mon Apr 5 21:14:10 UTC 2010 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Oct  5 11:26:52 2010 from localhost
help
Available commands:
(...)
| exit
| help [<command>]
| list
| quit
| request.alive
| request.all
| request.metadata <rid>
| request.on_air
| request.resolving
| request.trace <rid>
| uptime
| var.get <variable>
| var.list
| var.set <variable> = <value>
| version

Type "help <command>" for more information.
END
exit
Bye!
END
Connection to localhost closed.
```

This is an example of how you can use an existing secure access to secure the access to liquidsoap's command server. This way, you make sure that you are using a mainstream secure application, here SSH.

This example may be adapted similarly to use an online HTTP login mechanism, which is probably the most comment type of mechanism intented for the command line server.

CHAPTER 59

Streaming to Shoutcast

Although Liquidsoap is primarily aimed at streaming to Icecast servers (that provide much more features than Shoutcast), it is also able to stream to Shoutcast.

59.1 Shoutcast output

Shoutcast server accept streams encoded with the MP3 or AAC/AAC+ codec. You to compile Liquidsoap with `lame` support, so it can encode in MP3. Liquidsoap also has support for AAC+ encoding using FDK-AAC or using an external encoder. The recommended format is MP3.

Shoutcast output are done using the `output.shoutcast` operator with the appropriate parameters. An example is:

```
source = single("audiofile.ogg")

output.shoutcast(%mp3, host="shoutcast.example.org",
                       port=8000, password="changeme",
                       source)
```

As usual, `liquidsoap -h output.shoutcast` gives you the full list of options for this operator.

59.2 Shoutcast as relay

A side note for those of you who feel they ‘‘need” to use Shoutcast for non-technical reasons (such as their stream directory service...): you can still benefit from Icecast's power by streaming to an Icecast server, and then relaying it through a shoutcast server.

In order to do that, you have to alias the root mountpoint (“/”) to your MP3 mountpoint in your icecast server configuration, like this:

```
<alias source="/" dest="/mystream.mp3" />
```

Be careful that icecast often aliases the status page (`/status.xsl`) with the `/`. In this case, comment out the status page alias before inserting yours.

CHAPTER 60

Sources

Using liquidsoap is about writing a script describing how to build what you want. It is about building a stream using elementary streams and stream combinators, etc. Actually, it's a bit more than streams, we call them *sources*.

A source is a stream with metadata and track annotations. It is discretized as a stream of fixed-length buffers of media samples, the frames. Every frame may have metadata inserted at any point, independently of track boundaries. At every instant, a source can be asked to fill a frame of data.

The liquidsoap API provides plenty of functions for building sources. Some of those functions build elementary sources from scratch, others are operators that combine sources into more complex ones. An important class of sources is that of *active sources*, they are the sources that actively trigger the computation of your stream. Typically, active sources are built from output functions, because outputting a stream is the only reason why you want to compute it.

All sources, operators and outputs are listed in the scripting API reference.

60.1 How does it work?

To clarify the picture let's study in more details an example:

```
radio =
  output.icecast(
    %vorbis,mount="test.ogg",
    random(
      [ jingle ,
        fallback([ playlist1,playlist2,playlist3 ]) ]))
```

At every cycle of the clock, the output asks the `random` node for data, until it gets a full frame of raw audio. Then, it encodes the frame and sends it to the Icecast server. Suppose `random` has chosen the `fallback` node, and that only `playlist2` is available, and thus played. At every cycle, the buffer is passed from `random` to `fallback` and then to `playlist2` which fills it, returns it to `fallback` which returns it to `random` which returns it to the output.

At some point, `playlist2` ends a track. The fallback detects that on the returned buffer, and selects a new child for the next filling, depending on who's available. But it doesn't change the buffer, and returns it to `random`, which also

(randomly) selects a new child at this point, before returning the buffer to the output. On next filling, the route of the frame can be different.

Note that it is also possible to have the route changed inside a track, for example using the `track_sensitive` option of fallback, which is typically done for instant switches to live shows when they start.

The important point here is that **all of the above steps are local**. Everything takes place between one operator and its immediate children source; operators do not see beyond that point.

60.2 Fallibility

By default, liquidsoap outputs are meant to emit a stream without discontinuing. Since this stream is provided by the source passed to the output operator, it is the source responsibility to never fail. Liquidsoap has a mechanism to verify this, which helps you think of all possible failures, and prevent them. Elementary sources are either *fallible* or *infallible*, and this *liveness type* is propagated through operators to finally compute the type of any source. For example, a `fallback` or `random` source is infallible if an only if at least one of its children is infallible, and a `switch` is infallible if and only if it has one infallible child guarded by the trivial predicate `{ true }`.

On startup, each output checks the liveness type of its input source, and issues an error if it is fallible. The typical fix for such problems is to add one fallback to play a default file (`single()`) or a checked playlist (`playlist.safe()`) if the normal source fails. One can also use the `mksafe` operator that will insert silence during failures.

If you do not care about failures, you can pass the parameter `fallible=true` to most outputs. In that case, the output will accept a fallible source, and stop whenever the source fails, to restart when it is ready to emit a stream again.

60.3 Caching mode

In some situations, a source must take care of the consistency of its output. If it is asked twice to fill buffers during the same cycle, it should fill them with the same data. Suppose for example that a playlist is used by two outputs, and that it gives the first frame to the first output, the second frame to the second output: it would give the third frame to the first output during the second cycle, and the output will have missed one frame.

It is sometimes useful to keep this is mind to understand the behaviour of some complex scripts. The high-level picture is enough for users, more details follow for developers and curious readers.

The sources detect if they need to remember (cache) their previous output in order to replay it. To do that, clients of the source must register in advance. If two clients have registered, then caching should be enabled. Actually that's a bit more complicated, because of transitions. Obviously the sources which use a transition involving some other source must register to it, because they may eventually use it. But a jingle used in two transitions by the same switching operator doesn't need caching. The solution involves two kinds of registering: *dynamic* and *static activations*. Activations are associated with a path in the graph of sources' nesting. The dynamic activation is a pre-registration allowing a single real *static activation* to come later, possibly in the middle of a cycle. Two static activations trigger caching. The other reason for enabling caching is when there is one static activation and one dynamic activation which doesn't come from a prefix of the static activation's path. It means that the dynamic activation can yield at any moment to a static activation and that the source will be used by two sources at the same time.

60.4 Execution model

In your script you define a bunch of sources interacting together. Each source belongs to a clock, but clocks only have direct access to *active sources*, which are mostly outputs. At every cycle of the clock, active sources are animated: a chunk of stream (frame) is computed, and potentially outputted one way or another.

This streaming task is the most important and shouldn't be disturbed. Thus, other tasks are done in auxiliary threads: file download, audio validity checking, http polling, playlist reloading... No blocking or expensive call should be done in streaming threads. Remote files are completely downloaded to a local temporary file before use by the root thread. It also means that you shouldn't access NFS or any kind of falsely local files.

CHAPTER 61

Split and re-encode a CUE sheet.

CUE sheets are sometimes distributed along with a single audio file containing a whole CD. Liquidsoap can parse CUE sheets as playlists and use them in your request-based sources.

Here's for instance an example of a simple code to split a CUE sheet into several mp3 files with `id3v2` tags:

```
# Log to stdout
set("log.file",false)
set("log.stdout",true)
set("log.level",4)

# Initial playlist
cue = "/path/to/sheet.cue"

# Create a reloadable playlist with this CUE sheet.
# Tell liquidsoap to shutdown when we are done.
x = playlist.reloadable(cue, on_done=shutdown)

# We will never reload this playlist so we drop the first
# returned value:
s = snd(x)

# Add a cue_cut to cue-in/cue-out according to
# markers in "sheet.cue"
s = cue_cut(s)

# Shove all that to a output.file operator.
output.file(%mp3(id3v2=true,bitrate=320),
            fallible=true,
            reopen_on_metadata=true,
            "/path/to/$(track) - $(title).mp3",
            s)
```

CHAPTER 62

Stream contents

In liquidsoap, a stream may contain any number of audio, video and MIDI channels. As part of the type checking of your script, liquidsoap checks that you make a consistent use of stream contents, and also guesses what kind of stream your script is intended to work on. As with other inferred parameters, you do not necessarily need to read about stream contents typing if you're still learning the ropes of liquidsoap, but you might eventually need to know a little about it.

The content of a stream is described by the audio, video and MIDI arities. An arity might be fixed or variable. Fixed arities are usual natural numbers, described a number of channels that does change over time. For example, the stream type `(2,0,0)` describes streams that always have 2 audio channels and no channel of another type. Variable arities describes numbers of channels that vary over time. For example, the stream type `(*,0,0)` describes a stream which contains only audio, but whose number of channels might change at anytime – think of playing files, some of which being stereo, some mono, and some videos without any audio content. The stream type `(*+1,0,0)` also describes a variable number of audio channels, but with the guarantee that there will always be at least one.

In liquidsoap script language, there are three sorts of objects that rely on stream types: sources, requests and encoding formats. A source produces a stream, and it is important what kind of stream it produces when composing it with other sources. A request is an abstract notion of file, often meant to be decoded, and it is useful to know into what kind of stream it is meant to be decoded. Finally, a format describes how a stream should be encoded (*e.g.*, before output in a file or via icecast), and the stream content is also useful here for the format to make sense.

In this page, we explain how liquidsoap uses stream types to guess and check what you're doing.

62.1 Global parameters

You might have noticed that our description of stream contents is missing some information, such as sample rate, video size, etc. Indeed, that information is not part of the stream types, which is local to each source/request/format, but global in liquidsoap. You can change it using the `frame.audio/video.*` settings, shown here with their default values:

```
set("frame.audio.samplerate",44100)
set("frame.video.width",320)
set("frame.video.height",240)
set("frame.video.samplerate",25)
```

62.2 Checking stream contents

Checking the consistency of use of stream contents is done as part of type checking. There is not so much to say here, except that you have to read type errors. We present a few examples.

For example, if you try to send an ALSA input to a SDL input using `output.sdl(input.alsa())`, you'll get the following:

```
At line 1, char 22-23:
  this value has type
    source(audio=?A+1,video=0,midi=0)
    where ?A is a fixed arity type
  but it should be a subtype of
    source(audio=0,video=1,midi=0)
```

It means that a source with exactly onevideo channel was expected by the SDL output, but the ALSA output can only offer sources producing audio. By the way, `?A+1 where ?A is fixed` means that the ALSA input will accept to produce any number of channels, fixed once for all: it will attempt to initialize the soundcard with that number of channels and report a runtime error if that fails.

62.3 Conversions

The above example did not make much sense, but in some cases you'll get a type error on seemingly meaningful code, and you'll wonder how to fix it. Often, it suffices to perform a few explicit conversions.

Consider another example involving the SDL output, where we also try to use AO to output the audio content of a video: `liquidsoap output.ao(output.sdl(single("file.ogv")))`. This won't work, because the SDL output expects a pure video stream, but AO wants some audio. The solution is to split the stream in two, dropping the irrelevant content:

```
s = single("file.ogv")
output.sdl(drop_audio(s))
output.ao(drop_video(s))
```

Currently, the video dropping is useless because AO tolerates (and ignores) non-audio channels.

If you want to support both mono and stereo (and more) files within the same playlist, you'll need your `playlist` or `single` instance to have type `source(*+1,0,0)`. But this content type won't be supported by most operators, which require fixed arities. What you need to do is use `audio_to_stereo` which will normalize your variable arity audio into a fixed stereo audio.

The last conversion is muxing. It is useful to add audio/video channels to a pure video/audio stream. See `mux_video`, `mux_audio` and `mux_midi`.

62.4 Type annotations

You now have all the tools to write a correct script. But you might still be surprised by what stream content liquidsoap guesses you want to use. This is very important, because even if liquidsoap finds a type for which it accepts to run, it might not run as you intend: a different type might mean a different behavior (not the intended number of audio channels, no video, etc).

Before reading on how liquidsoap performs this inference, you can already work your way to the intended type by using type annotations.

For example, with `output.alsa(input.alsa())`, you'll see that liquidsoap decides that stereo audio should be used, and consequently the ALSA I/O will be initialized with two channels. If you want to use a different number of channels, for example mono, you can explicitly specify it using:

```
output.alsa((input.alsa():source(1,0,0)))
```

62.5 Guessing stream contents

When all other methods fail, you might need to understand a little more how liquidsoap guesses what stream contents should be used for each source.

First, liquidsoap guesses as much as possible (without making unnecessary assumption) from what's been given in the script. Usually, the outputs pretty much determine what sources should contain. A critical ingredient here is often the encoding format. For example, in

```
output.icecast(%vorbis,mount="some.ogg",s)
```

`%vorbis` has type `format(2,0,0)`, hence `s` should have type `source(2,0,0)`. This works in more complex examples, when the types are guessed successively for several intermediate operators.

After this first phase, it is possible that some contents are still undetermined. For example in `output.alsa(input.alsa())`, any number of audio channels could work, and nothing helps us determine what is intended. At this point, the default numbers of channels are used. They are given by the setting `frame.audio/video/midi.channels` (whose defaults are respectively `2`, `0` and `0`). In our example, stereo audio would be chosen.

---

header-includes: | \DeclareUnicodeCharacter{03C0}{π} ... Basically streaming videos does not change anything compared to streaming audio: you just have to use video files instead of sound files! For instance, if you want to stream a single file to an icecast server in ogg format (with theora and vorbis as codecs for audio and video) you can simply type:

```
source = single("video.avi")

output.icecast(
        %ogg(%theora(quality=25,width=320,height=240),%vorbis),
        host="localhost",
        port=8000,
        password="hackme",
        mount="/videostream",
        source)
```

And of course you could have used a `playlist` instead of `single` to have multiple files, or used other formats for the stream.

In order to test a video stream, it is often convenient to use the `output.sdl` operator (or `output.graphics`) which will open a window and display the video stream inside. These can handle streams with video only, you can use the `drop_audio` operator to remove the sound part of a stream if needed.

You should be expecting much higher resource needs (in cpu time in particular) for video than for audio. So, be prepared to hear the fan of your computer! The size of videos have a great impact on computations; if your machine cannot handle a stream (i.e. it's always catching up) you can try to encode to smaller videos for a start.

CHAPTER 63

Useful tips & tricks

Video is a really exciting world where there are lots of cool stuff to do.

63.1 Transitions

Transitions at the beginning or at the end of video can be achieved using `video.fade.in` and `video.fade.out`. For instance, fading at the beginning of videos is done by

```
source = video.fade.in(transition="fade",duration=3.,source)
```

63.2 Adding a logo

You can add a logo (any image) using the `video.add_image` operator, as follows:

```
source = video.add_image(
       width=30,height=30,
       x=10,y=10,
       file="logo.jpg",
       source)
```

63.3 Inputting from a webcam

If your computer has a webcam, it can be used as a source thanks to the `input.v4l2` operator. For instance:

```
output.sdl(input.v4l2())
```

63.4 Video in video

Suppose that you have two video sources `source` and `source2` and you want to display a small copy of `source2` on top of `source`. This can be achieved by

```
source2 = video.scale(scale=0.2,x=10,y=10,source2)
source = add([source,source2])
```

63.5 Scrolling text

Adding scrolling text at the bottom of your video is as easy as

```
source = video.add_text.sdl(
       font="/usr/share/fonts/truetype/ttf-dejavu/DejaVuSans.ttf",
       "Hello world!", source)
```

You might need to change the `font` parameter so that it matches a font file present on your system.

63.6 Effects

There are many of effects that you can use to add some fun to your videos: `video.greyscale`, `video.sepia`, `video.lomo`, etc. Read the documentation to find out about them. If you have compiled Liquidsoap with frei0r support, and have installed frei0r plugins, they will be named `video.frei0r.*`. You can have a list of those supported on your installation as usual, using `liquidsoap --list-plugins`.

63.7 Presenting weather forecast

You can say that a specific color should be transparent using `video.transparent`. For instance, you can put yourself in front of a blue screen (whose RGB color should be around 0x0000ff) and replace the blue screen by an image of the weather using

```
img = single("weather.jpg")
cam = input.v4l2()
cam = video.transparent(color=0x0000ff,precision=0.2,cam)
source = add([img,cam])
```

CHAPTER 64

Detailed examples

64.1 The anonymizer

Let's design an ''anonymizer" effect: I want to blur my face and change my voice so that nobody will recognise me in the street after seeing the youtube video. Here is what we are going to achieve:

```
# Input from webcam
cam = input.v4l2()

# Detect faces (this generates a white disk over faces)
mask = video.frei0r.opencvfacedetect(cam)
# Pixellize the video
censored = video.frei0r.pixeliz0r(blocksizex=0.1,blocksizey=0.1,cam)
# Generate a mask for video without the face
unmask = video.frei0r.invert0r(mask)
# Put the pixellized face over the video
s = video.frei0r.addition(
  video.frei0r.multiply(mask,censored),
  video.frei0r.multiply(unmask,cam))
# We have to bufferize the source because its clock it GStreamer's clock
s = buffer(buffer=0.1,mksafe(s))

# Input audio from microphone
mic = input.pulseaudio(clock_safe=false)
# Transpose sound to generate a funny voice
mic = soundtouch(pitch=1.5,mic)
# Add sound to video
s = mux_audio(audio=mic,s)

# Let's hear the sound
output.pulseaudio(fallible=true,s)
# Let's see the video
output.sdl(fallible=true,drop_audio(s))
```

(continues on next page)

(continued from previous page)

```
s = mksafe(s)
# Output the video/sound into a file in theora/vorbis format
output.file(%ogg(%theora(quality=63),%vorbis), "anonymous.ogv", s)
```

64.2 Controlling with OSC

In this example we are going to use OSC integration in order to modify the parameters in realtime. There are many OSC clients around, for instance I used TouchOSC :

```
# Set the OSC port to match TouchOSC's default port
set("osc.port",8000)

# Input from the webcam
s = input.v4l2_with_audio()
s = mksafe(s)

# We get the angle from fader 3
angle = osc.float("/1/fader3", 0.)
# we rescale the position of fader 3 so that it corresponds to a 2π rotation
angle = fun() -> angle() * 3.1416 * 2.
# ...and we rotate the video according to the angle
s = video.rotate(speed=0.,angle=angle,s)
# Change brightness according to fader 1
s = video.frei0r.brightness(brightness=osc.float("/1/fader1",0.5),s)
# Change contrast according to fader 2
s = video.frei0r.contrast0r(contrast=osc.float("/1/fader2",0.5),s)

# We have to buffer here otherwise we get clocks problems
s = buffer(s)

# Output sound and video
output.pulseaudio(fallible=true,s)
output.sdl(fallible=true,drop_audio(s))
```

64.3 Blue screen

You want to show yourself in front of a video of a bunny, as in

```
# The video of the bunny
s = single("big_buck_bunny_720p_stereo.ogg")
# Input from the webcam
cam = input.v4l2()
# Flip the video around a vertical axis so that it is easier
# to position yourself
cam = video.frei0r.flippo(x_axis=true,cam)
# Make the white background transparent
# I had to tweak the precision parameter so that I will be seen
# but not the wall
cam = video.transparent(color=0xffffff,precision=0.64,cam)
# Superpose the two videos
s = add([s,cam])
```

(continues on next page)

(continued from previous page)

```
# Output to SDL
output.sdl(fallible=true,drop_audio(s))
```

64.4 Encoding with GStreamer codecs

Gstreamer codecs can be used to encode videos and audio as any natively supported format. For instance, suppose that you want to stream using harbor in x264 / mp3. This can be achieved as follows:

```
# Set the values for video size and fps.
# On my standard computer, higher values means
# that we cannot encode in realtime.
set("frame.video.width", 320)
set("frame.video.height",240)
set("frame.video.samplerate",12)

# The video we want to stream.
s = single("big_buck_bunny_720p_stereo.ogg")
# This hack is necessary (for now) in order
# to leave the synchronization to GStreamer.
clock.assign_new(sync=false,[s])

output.harbor(
  format="video/mpeg",
  icy_metadata="false",
  mount="/test",
  %gstreamer(video="x264enc speed-preset=1",audio="lamemp3enc"),
  s)
```

The video can be read after that at http://localhost:8000/test and of course an `output.icecast` or `output.file` could have been used instead of `output.harbor` depending on your needs.

64.5 Streaming with GStreamer

The usual way to stream a video is using icecast, as for audio. However, it can happen that you want to use weired formats or ways to to stream. In this case, using GStreamer as output (as opposed to simply a codec as above) might be a good idea. For instance, suppose that you want to stream mp4 video using RTP. This can be done as follows:

```
s = single("test.mp4")
output.gstreamer.video(pipeline="videoconvert ! avenc_mpeg4 ! rtpmp4vpay config-
↪interval=2 ! udpsink host=127.0.0.1 port=5000", s)
```

The stream can then be read with vlc for instance using `vlc test.sdp`. Here, the contents of the file `test.sdp` is

```
v=0
m=video 5000 RTP/AVP 96
c=IN IP4 127.0.0.1
a=rtpmap:96 MP4V-ES/90000
```

CHAPTER 65

Frequently asked questions

65.1 audio=1+_

When I try

```
s = input.v4l2_with_audio()
output.sdl(s)
```

I get the error

```
At line 2, char 13:
  this value has type
    active_source(audio=1+_,...) (inferred at ../scripts/gstreamer.liq, line 20, char␣
↪30-121)
  but it should be a subtype of
    active_source(audio=0,...)
```

This error means that the stream `s` has an audio channel (as indicated by `audio=1+_`) whereas `output.sdl` wants no audio channel. Namely, it's type is

```
$ liquidsoap -h output.sdl

Display a video using SDL.

Type: (?id:string,?fallible:bool,?on_start:(()->unit),
 ?on_stop:(()->unit),?start:bool,
 source(audio=0,video=1,midi=0))->
active_source(audio=0,video=1,midi=0)
```

which means that it wants 0 audio channel, 1 video channel and 0 midi channel. The solution to correct the script is simply to remove the audio channel using the `drop_audio` operator:

```
s = input.v4l2_with_audio()
output.sdl(drop_audio(s))
```

CHAPTER 66

Advanced parameters

66.1 Default size for videos

Internally, Liquidsoap uses a video format which is the same for all frames. You can change it by doing

```
set("frame.video.width",320)
set("frame.video.height",240)
set("frame.video.samplerate",24)
```

Using higher values result in higher quality videos produced, but this also means more computations to perform!

66.2 Converters

Most videos need to be rescaled to the Liquidsoap internal format. The default converter is the GAVL library but you can choose other (such as `native` or `ffmpeg`) by

```
set("video.converter.preferred", "ffmpeg")
```

If you are using `gavl`, you can change the scaling mode by

```
set("video.converter.gavl.scale_mode", "quadratic")
```

Several modes beside `quadratic` are available, use `liquidsoap --conf-descr` to discover them. Keep in mind that you should keep a good balance between performance and quality!

CHAPTER 67

A simple video script

The other day, I wanted to prepare some videos of my favorite reggae and soul tunes for uploading them to YouTube. My goal was very simple: prepare a video with the music, and a static image.

After briefly digging for a simple software to do that, which I could not find, I said ‘‘hey, why not doing it with liquidsoap”? Well, that is fairly easy!

Here is the code:

```
# Log to stdout
set("log.file",false)
set("log.stdout",true)
set("log.level",4)
# Enable video
set("frame.video.width",640)
set("frame.video.height",480)

audio_file = "/tmp/bla.mp3"
video_file = "/tmp/bla.jpg"

# Grab file's title
r = request.create(audio_file)
title =
  if request.resolve(r) then
    meta = request.metadata(r)
    meta["title"]
  else
    # File not readable
    log("Error: cannot decode audio file!")
    shutdown ()
    ""
  end
title =
  if title == "" then
     "Unknow title"
  else
```

(continues on next page)

(continued from previous page)

```
    title
  end

# The audio song.
audio = request.queue(interactive=false,queue=[r])

# Create a video source with the image for video track
video = single(video_file)

# Mux audio and video
#source = mux_audio(audio=audio,video)
source = mux_video(video=video,audio)

# Disable real-time processing, to process with the maximun speed
source = clock(sync=false,source)

# Output to a theora file, shutdown on stop
output.file(%ogg(%vorbis,%theora),
            id="youtube",fallible=true,
            on_stop=shutdown,reopen_on_metadata=true,
            "/tmp/#{title}.ogv",
            source)
```

This should produce on file named `<title>.ogv` where `<title>` is the title metadata of your song.

Inspired from blog.rastageeks.org.

CHAPTER 68

Indices and tables

- genindex
- modindex
- search