Liquidsoap manual

David Baelde and Samuel Mimram

October 15, 2007

Contents

Chapter 1

Liquidsoap

1.1 Liquidsoap

Liquidsoap is a powerful tool for building complex audio streaming systems, typically targetting internet radios. 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 streamer you want. This design makes liquidsoap flexible and easily extensible.

We believe that liquidsoap is easy to use. For basic uses, the scripts simply consists of the definition of a tree of sources. It is good to use liquidsoap even for simple streams which could be produced by other tools, because it is extensible: when you want to make your stream more complex, you are still able to stay in the same framework, and your script will remain maintainable. Of course, this will require at some point a deeper understanding of liquidsoap and its scripting language.

If you're new to liquidsoap, you'd probably like to read about the installation procedure and take the quickstart tour. Then you may also enjoy to learn more about the main concepts on which liquidsoap is built. When you'll master these concepts, you'll only need to take a look at the reference (scripting language, API and settings) and get a few ideas from the recipes to be able to design whatever stream you need. Finally, have a look at the telnet tutorial to find out how to interact in various ways with a running liquidsoap.

Liquidsoap is written in OCaml and is part of the savonet project.

Acknowledgement for the readers of the PDF version. The file you're reading has been automatically generated from savonet's wiki. It can be useful to get directly there, in particular if you need to copy a code snippet: `http://savonet.sf.net/wiki/Liquidsoap`.

Acknowledgement for the Wiki readers. There is a PDF file automatically generated from selected pages of this wiki. It can be useful for printing, and is available in liquidsoap distribution.

1.1.1 Features

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

- Playing from files, playlists, 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, etc.
- Queuing of user requests.
- Supports arbitrary transitions: you can have fade, cross-fade, jingle insertion, etc.

- Per-track settings of transitions via metadatas `liq_fade_in`, `liq_fade_out`, `liq_start_next`, `liq_append` and `liq_prepend`.
- Input of other Icecast streams (Ogg/Vorbis or MP3): useful for switching to a live show.
- Blank detection.
- Definable event handlers on new tracks and excessive blank.
- Metadata rewriting.
- 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 and peercast (mp3/ogg) or a local file (wav/mp3/ogg).
- Output to speakers using libao.
- Output to ALSA speaker, input from ALSA microphone. There are some unfixed issues there.
- Distributed encoding using RTP (but it's unmaintained and experimental!)
- Arbitrary mixing of several sources together.
- Interactive control of many operators via telnet, or indirectly using perl/python scripts, pyGtk GUI, web/irc interfaces (not released, mail us)...
- Speech and sound synthesis.

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.

1.1.2 Non-Features

Liquidsoap is a flexible tool for generating audio 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 is a graphical application using the Python module for controlling a running liquidsoap: liguidsoap.

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.

1.1.3 History

- 0.3.0 brought a lot of improvements at all levels, polishing, uniformization and documentation.
- 0.2.0 was the first working release, not so unstable and already featuring the main good ideas.
- 0.1.0 was the release we had to do at the end of the academic project, not working at all :p

1.2 Installation

Several ways of installing liquidsoap are possible. In most cases you can choose which features you want. Here are liquidsoap's dependencies (all OCaml libraries are distributed by Savonet, except Camomile):

- ocamlfind (http://www.ocaml-programming.de/programming/findlib.html)
- ocaml-dtools
- ocaml-vorbis
- ocaml-shout

And also optional dependencies:

- ocaml-mad for mp3 decoding
- libid3tag (http://www.underbit.com/products/mad/) for reading mp3's id3 metadata
- ocaml-mp3id3 for reading mp3's id3 metadata
- camomile (http://camomile.sourceforge.net/) for detecting metadata encodings and re-encoding them to utf8
- ocaml-lame for mp3 encoding
- ocaml-alsa for alsa input/output
- libortp (http://www.linphone.org/) for RTP input/output
- wget (http://www.gnu.org/software/wget/) for downloading remote files (http, https, ftp)
- ufetch (provided by ocaml-fetch) for downloading remote files (smb, http, ftp)
- festival (http://www.cstr.ed.ac.uk/projects/festival/) for speech synthesis (say)

1.2.1 Install from source tarballs

The primary mean of stable distribution is source tarballs. They are available on the download section (http://sourceforge.net/project/showfiles.php?group_id=89802) of the project's page on sourceforge. They all follow the GNU conventions, and are built and installed using the common `./configure`, `make` and `make install`.

1.2.2 Subversion repository (and other distributions)

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

```
svn co https://svn.sourceforge.net/svnroot/savonet/trunk savonet
```

From every sub-project's directory you can build and install the package using `./bootstrap`, `./configure`, `make` and optionally `make install`.

From the toplevel `savonet` directory you can also directly build a vanilla liquidsoap. It's fast and doesn't require you to install the libraries. The steps to follow are simple:

```
# Edit PACKAGES to choose which feature you want
./bootstrap
./configure
make
# To install liquidsoap, you'll usually need to type the following as root
make install
```

1.2.3 Debian

Debian packages are available for some libraries on the download section (http://sourceforge.net/project/showfiles.php?group_id=89802) of the project's page on sourceforge. A `.deb` package, for example `toto.deb`, can be installed using `dpkg -i toto.deb`.

The main libraries already entered the official Debian distribution, and we're currently working on packaging liquidsoap too.

1.2.4 Gentoo

The ebuilds available on our SVN are very outdated, but there is ongoing work on updating and including them directly in the official distribution.

1.2.5 OSX

There have been successful installations on OSX (both Intel and PPC), using Fink and the Godi distribution of OCaml. Claudio reports his two successes on the OSX page (http://savonet.sf.net/wiki/InstallationOSX).

1.3 Quickstart

1.3.1 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 (http://www.icecast.org/ices.php), ...) which creates an audio stream (ogg or mp3);
- the streaming media server (icecast (http://www.icecast.org), shoutcast, ...) which relays several streams from their sources to their listeners;
- the media player (xmms, winamp, ...) which gets the audio stream from the streaming media server and plays it to the listener's speakers.

The stream is always passed from the source 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.

A source client 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.

You may have an already setup icecast server. Otherwise you can start the tutorial using liquidsoap's output to speakers, or immediately install and configure your 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.

You'll have to provide liquidsoap with information about the icecast server, unless the default values are OK: host (defaults to `localhost`), port (defaults to `8000`, which is icecast's default) and source password (defaults to icecast's default `hackme`).

Now, let's create that audio stream.

1.3.2 Starting to use liquidsoap

Liquidsoap is a script language. To use it, you have to write a script in some file, say `myscript.liq`, and then run `liquidsoap myscript.liq`. You can also put `#!/path/to/your/liquidsoap` as the first line of your script, you'll then be able to run it by simply doing `./myscript.liq`. Usually, the path of the `liquidsoap` executable is `/usr/bin/liquidsoap`, and we'll use this in the following.

Liquidsoap will write log messages in a file, and possibly in the console, depending on the settings preamble of your script. In this tutorial, we'll always use the same settings, for simplicity:

```
#!/usr/bin/liquidsoap

# Put the log file in some directory where you have the write permission
set log.dir = "/tmp"

# Print log messages to the console
set log.stdout = true
# This can also be done by passing the -v option to liquidsoap
```

Lines starting with `#` are ignored, they are just comments. Those who know ruby will note that this is not the only similarity with it's syntax.

A simple example

In the first example we'll play a playlist. Let's put a list of audio files in `playlist.pls`: one file per line, lines starting with a `#` are ignored. The following liquidsoap script plays that list in shuffle mode:

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

set log.dir = "/tmp"

output = ouput.ao
# Output via libao is the most portable and stable option.
# You may also be able to use output.alsa.
# If you have a configured icecast server,
# uncomment the next two lines and edit the parameters
# output = output.icecast(mount="foo.ogg",host="localhost",
#                         port=8000,password="hackme")

output(playlist.safe("playlist.pls"))
```

Hopefully, you're now listening to your playlist, via icecast or not. You may have noticed that liquidsoap's startup is quite slow, especially if you have a large playlist. That's because we used the `playlist.safe` source, which checks every file in the playlist and removes those which cannot be decoded – and fails if none can. These checks are here because liquidsoap cares about your stream and makes sure that it'll never stop. Later in the tutorial we explain how to get safe sources without using the heavy `playlist.safe` sources but a mix of unsafe `playlist` sources and other safe sources.

In a liquidsoap script, you build source objects. Liquidsoap provides many functions for creating sources from scratch (e.g. `playlist`) and creating complex sources by wrapping 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, icecast, etc. These active sources are the root of a liquidsoap instance, the sources which bring life into it.

A complex example

Now for a more complex example. The following script:

- sets up several outputs;
- plays different playlists during the day;
- inserts about 1 jingle every 5 songs;
- adds one special jingle at the beginning of every hour on top of the normal stream;
- plays user requests – done via the telnet server;
- and relays live shows, inserting jingles as transitions between live shows and the usual program.

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

set log.dir = "/tmp"

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

base = "~/radio/"

default = single(base ^ "default.ogg")

day     = playlist(base ^ "day.pls")
night   = playlist(base ^ "night.pls")
jingles = playlist(base ^ "jingles.pls")

clock   = single(base ^ "clock.ogg")
start   = single(base ^ "live_start.ogg")
stop    = single(base ^ "live_stop.ogg")

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

# Add the ability to relay live shows
# We first define the transition function
def transition(jingle,a,b)
  add(normalize=false,
          [ fade.initial(b),
            sequence(merge=true,
                     [blank(duration=1.),jingle,fallback([])]),
            fade.final(a) ])
end
```

```
full =
  fallback(track_sensitive=false,
               transitions=[ transition(start), transition(stop) ],
               [input.http("http://localhost:8000/live.ogg"),
                radio])

# Output the full stream in OGG and MP3
output.icecast.mp3(mount="radio",full)
output.icecast(mount="radio.ogg",full)

# Output the stream without live in OGG
output.icecast(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 the first example. To start a real live show you can use darkice, or simply liquidsoap if you have a working ALSA input, with the following script:

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

set log.dir = "/tmp"

# Get your microphone output and send it to icecast
# Add other icecast parameters if needed
output.icecast(mount="live.ogg",input.alsa())
```

Interaction with the server

In the previous examples we set up a `request.queue` source to play user requests. To push requests in that queue you need to interact with the telnet server, which also provides many other services. By default it is only accessible from the host where liquidsoap runs. You can learn more on that topic with the [[LiqTelnetTuto telnet tutorial] and settings description. Here is a sample session:

```
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
[...]
END
request.push http://remote/audio.ogg
6
END
trace 6
[...]
END
help
[...]
```

```
END
exit
```

Of course, telnet isn't user friendly. But it is easy to write scripts to interact with liquidsoap in that way. Examples of such tools are liguidsoap and bottle.

1.3.3 That source is fallible??!

If you start trying your own examples, you will probably quickly run into that error. In liquidsoap, we say that a source is infallible if it will be always available: it will always be able to fill some audio frames. Otherwise, it is fallible. Liquidsoap checks that the child of an output is infallible, so that you can trust your output. Basically, it helps you to remember to always put some trusted file somewhere in the program.

For example `request.queue()` is not available when there is no user request, so it is fallible. A `playlist` might become temporarily unavailable if it encounters an undecodable file, or if it needs to download files on an unreliable network. But `playlist.safe("my.pls")` is always available if `my.pls` is a valid file containing a playlist of valid local files – because all files are checked in advance.

The typical way to turn a fallible source `src` into an infallible one is to pick a default valid local file `failure.ogg` and use `fallback([src,single("failure.ogg")])`. It will never fail, because if your initial source fails, the default local file will always be there to fill the stream.

Use `liquidsoap --check my.liq` for checking a file without running it, including fallibility checks.

1.3.4 Daemon mode

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 serious radio.

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

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

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

1.3.5 What's next?

You should now learn more about liquidsoap's scripting language. Once you'll know the syntax and types, you'll probably need to refer to the scripting reference and the settings reference, or see examples. For a better understanding of liquidsoap, it is also suggested to read more about the concepts of the system.

1.4 Concepts

1.4.1 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 – in liquidsoap's code there is a `Types.source` type, and in `*.liq` scripts one of the elementary datatypes is source.

A source is a stream with metadata and track annotations. It is discretized as a stream of fixed-length buffers of raw audio, the frames. Every frame may have metadata inserted at any point, independantly of track boundaries. At every instant, a source can be asked to fill a frame of data. Track boundaries are denoted by a single denial of completely filling a frame. More than one denial is taken as a failure, and liquidsoap chooses to crash in that case.

To build sources in liquidsoap scripts, you need to call functions which return type is `source`. For convenience, we categorize these functions into three classes. The *sources* (sorry for redundancy, poor historical reasons) are functions which don't need a source argument – we might call them elementary sources. The *operators* need at least one source argument – they're more about stream combination or manipulation. Finally, some of these are called *outputs*, because they are active operators (or active sources in a few cases): at every instant they will fill their buffer and do something with it. Other sources just wait to be asked (indirectly or not) by an output to fill some frame.

All sources, operators and outputs are listed in the scripting API reference.

How does it work?

To clarify the picture let's study in more details an example.

```
radio = output.icecast(
        mount="test.ogg",
        random([ jingle , fallback([ playlist1 , playlist2 , playlist3 ])])
        )
```

At every tick, the output asks the "random" node for data, until it gets a full frame of raw audio. Then it encodes it, 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 tick, 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. Every step is local.

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 selects a new child, randomly, and return the buffer to the output. On next filling, the route of the frame can be different.

It is 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.

Fallibility

Outputs expect their input source to never fail, so that the stream never ends. Liquidsoap has a mechanism to verify this, and 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. A `fallback` or `random` source is infallible if and only if at least one of their children is infallible. A `switch` is infallible if and only if it has one infallible child guarded by the trivial predicate `true` . And so on.

On startup, outputs will check the liveness type of their input sources, and you'll get an error if one of these is fallible. The common answer to such errors is op add one fallback to play a default file or a checked playlist (`playlist.safe`) if the normal source fails. Often, the error is excessive, it simply means that your (unchecked) playlists could all be corrupted, which is unlikely. But sometimes, it also helps one to avoid the case where a playlist fails because it spent too much time trying to download remote files.

Caching mode

In some situations, a source must take care about the consistency of its output. If it is asked twice to fill buffers during the same time tick, it should fill them with the same data. Suppose

for example that a playlist is listened 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 tick, and the output will have missed one frame.

Keeping that in mind is required 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 the 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 real *static activation* to come later, possibly in the middle of a time tick, from a super-path – *i.e.* a path of which the first one is a prefix. 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.

1.4.2 Execution model

In your script you define a bunch of sources interacting together. The output sources hook their output function to the root thread manager. Then the streaming starts. At every tick the root thread calls the output hooks, and the outputs do their jobs. This 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 the root thread. 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.

1.4.3 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")`.

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, if the request is an audio one, liquidsoap check that the file is decodable: there should be a valid decoder for it (this isn't based on the extension but on the success of a format decoder), the decoder shouldn't yield an empty stream, and opening the decoder should be fast (less than 0.5 seconds), so that the opening of the audio file for its real playing in the main thread doesn't freeze it for too long.

Currently supported protocols

- SMB, FTP and HTTP using ufetch (provided by our ocaml-fetch distribution)
- HTTP, HTTPS, FTP thanks to wget
- 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 @, and you're still listening to Geek Radio.`
- ANNOTATE for manually setting metadata, typically used in `annotate:nick="vodka-goo",media=irc,message="special for sam":ftp://bla/bla/bla`. The extra metadata can then be synthetized 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.

Currently supported formats

- MP3
- Ogg/Vorbis
- WAV

1.5 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. To fit its particular purpose, it has first-class sources and requests (see liquidsoap's concepts) and a syntax extension for simply specifying time intervals.

A liquidsoap script starts with a settings section. Then comes a sequence of expressions, where you mostly define sources and active sources, which will animate your stream.

1.5.1 Constants

Constants are used in the settings preamble and in the body of a script. Their syntax is 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.

1.5.2 Settings

That part of the script allows you to control the global behaviour of liquidsoap: log file, log levels, daemon mode... by defining some of the available settings.

The settings preamble is very simple: a sequence of `set VAR = CONST`. `CONST` must be a constant of type bool, int or string as described earlier, or a list of strings noted as `[ "foo", 'bar' ]`. More complex expressions are not allowed here.

The type of a setting is very important: the `int` setting "bla" and the `bool` setting "bla" are distinct and can coexist; so if you put the wrong type for a configuration variable, it will simply be ignored by liquidsoap which will lookup only for the expected type.

1.5.3 Expressions

You can form expressions by using:

- Constants and variable identifiers. Identifier are made of alphanumerics, underscore and dot: `[a-zA-Z0-9_\.]*`
- Lists and tuples: `[expr,expr,...]` and `(expr,expr,..)`
- Sequencing: expressions may be sequenced, just juxtapose them. Usually one puts one expression per line. Optionally, they can be separated by a semicolon. The type of a sequence of expressions is the type of the last expression – just as a sequence evaluates to its last expression.
- Application `f(x,y)` of arguments to a function. Application of labeled parameters is as follows: `f(x,foo=1,y,bar="baz")`. The relative order of two parameters doesn't matter as long as they have different labels.
- Definitions using def-end: `def source(x) = s = wrap1(x) ; wrap2(s) end` or `def pi = 3.14 end`. The `=` is optional, you may prefer multi-line definitions without it. The definition of a function with two named parameters, the second one being optional with default value `13` is as follows: `def f(~foo,~bar=13) = body end`.
- Shorter definitions using the equality: `pi = 3.14`. This is never an assignment, only a new local definition!
- Anonymous functions: `fun (arglist) -> expr`. Don't forget to use parenthesis if you need more than one expr: `fun (x) -> f1(x) ; f2(x)` will be read as `(fun (x) -> f1(x)) ; f2(x)` not as `fun (x) -> (f1(x) ; f2(x))`.
- Code blocks: `expr` is a shortcut for `fun () -> expr`.

No assignation, 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 its body where parameters have been substituted. 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.

```
# return type of foo will be string.
def foo ()
        a = bar()
```

```
        b = 1
        "string"
end
```

Type of an application. The type of an application is the return type of function if all mandatory arguments are applied:

```
def foo ()
       1
end

# a will be an integer
a = foo()
```

Otherwise, the application is "partial", and the expression has the type of a function (see below for more about partial applications).

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(host="streamer",port="8080",password="sesame")
```

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.

1.5.4 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`, `string`, but also `source` and `request`. 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)]`.

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.initial(to) ])
```

1.5.5 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 and all components `?x` are optional. 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.

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`.

It is possible to forget the `m` for minutes if hours are specified – and seconds unspecified, obviously.

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)
])
```

1.6 Liquidsoap settings

Liquidsoap scripts start with a settings section, for defining a few global variables affecting the behaviour of the application. The settings are typed, and can be `string`, `int`, `bool` or `string list`.

Below is a presentation of available settings. Every section starts with a list of settings, together with their types. Then comes the description of the group of settings.

1.6.1 Logging

```
string log.dir
string log.file
bool   log.stdout
int    log.level
int    log.level.[label]
```

When executing `[script].liq`, liquidsoap logs some information in `[log.dir]/[script].log`. The default value for `[log.dir]` is set during configuration, typically `/var/log/liquidsoap`. It can also be set using the `string` setting `log.dir`. You can also override the filename, using the `log.file` setting.

It is often useful while debugging to directly see on the log on the standard output. The `log.stdout` can be set to `true` to achieve that.

Finally you can tweak the amount of information by changing the log levels. The higher it is, the less info you get. Log level 1 is for errors, 2 for warnings, 3 for information, 4 for annoying information, etc. `log.level` is the default level, you can also filter your logs more specifically using the label-specific `log.level.[label]` settings.

1.6.2 Daemon

```
bool   daemon
string daemon.piddir
string daemon.piddfile
```

Liquidsoap can detach and run as a daemon, if the `daemon` flag is set. In that case it will put the PID of the daemon process in `[daemon.piddir]/[script].pid`. The default piddir is typically `/var/run/liquidsoap`. As for the log file, the PID file can be overriden using the `daemon.pidfile` setting.

1.6.3 Server

```
bool server
int  server.port
bool server.public
```

You can interact with an instance of liquidsoap thanks to a simple telnet interface, where you can get info and control your sources. By default, that service is ran on port `1234` and is only available from the local host, since it currently doesn't support any kind of access control or authentification. You can disable it, change the port and the availability to the full network thanks to these settings. For GeekRadio, the service is public in order to be available for an IRC bot running on another server, but it is then restricted to the local network by the firewall.

1.6.4 Misc

```
string list tag.encodings
int   max_rid
float decoding.buffer.length
float max_latency
```

If your liquidsoap has charset reencoding support (using Camomile), then `tag.encodings` is used to guess the encoding of strings. The default is `["UTF-8","ISO-8859-1"]`.

The next settings are really low-level, and one should rarely need to change them.

The `max_rid` allows you to tune the maximum request ID. Making it a bit larger allows to access requests a while after that they are destroyed, to read the request trace or metadata. On the other hand, having too many requests can be messy and a bit heavy.

For some features (typically `fade.out()` and `cross()`) Liquidsoap needs to know precisely the remaining time in a track. It only roughly evaluates it at the beginning of a file, but it has to be precise at the end. This is done by decoding data in advance in a buffer of duration `decoding.buffer.length` seconds (defaults to 10). In a nutshell, if you use cross-fadings or fade-outs with a maximum duration of D, you should set the buffer's length to D too for perfect precision. However, it has a little cost on opening of a file while the buffer is filled, so be careful if you're short of computing power.

Sometimes, liquidsoap may get a bit late, in which case it'll run faster for a while to fill the gap. But when the latency is higher than `max_latency` seconds (defaults to 60), it'll restart all the outputs in order to cancel the latency. It shouldn't happen much, but it's there just in case.

1.7 Cookbook

The recipes show how to build a source with a particular feature. To try it, turn that into a script which plays the source directly to your speaker:

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

set log.dir = "/tmp"
set log.stdout = true

recipe = # <fill this>
output.ao(recipe)
```

```
# Output via libAO is the most stable and portable operator.
# You can also output to speakers via ALSA, or to a file, icecast, etc.
```

1.7.1 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 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.

1.7.2 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) ])
```

1.7.3 Fancy effects

```
# 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.

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

1.7.4 Unix interface, dynamic requests

`request.dynamic` is a source which takes a custom function for creating its new requests. This function can be used to call an external program. The source expects a `()->request` function. To create the request, the function will have to use the `request` 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 concepts page), 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.

The simplest example takes the output of an external script as an URI to create a new request:

```
request.dynamic({ request(get_process_output("my_script my_params")) })
```

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

```
request.dynamic({ request("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="seeed"`, liquidsoap will call the function, which will call `bubble-query 'artist="seed"'` which will output 10 lines, one URI per line.

1.7.5 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.

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. Other possible transition functions:

```
# A simple (long) cross-fade
def crossfade(a,b)
  add(normalize=false,
          [ sequence([ blank(duration=5.),
                       fade.initial(duration=10.,b) ]),
            fade.final(duration=10.,a) ])
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.
def next(j,a,b)
  add(normalize=false,
```

```
          [ sequence(merge=true,
                     [ blank(duration=3.),
                       fade.initial(duration=6.,b) ]),
            sequence([fade.final(duration=9.,a),
                      j,fallback([])]) ])
end

# A similar transition, which does a cross-fading from A to B
# and adds a jingle
def transition(j,a,b)
  add(normalize=false,
          [ fade.initial(b),
            sequence(merge=true,
                     [blank(duration=1.),j,fallback([])]),
            fade.final(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") ])
```

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

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`.

Chapter 2

Advanced topics

2.1 Using the telnet interface

Tobias Luther was kind enough to write this tutorial about the use of liquidsoap's telnet interface. In this example there are two queues for requests, one with the id `request`, one with the id `scheduler`. There are also two switched playlists, one for daytime, one for nighttime and a playlist for jingles. Here's a list of commands with examples of their use in this case.

Abbreviations:

- `<rid>` = request id
- `<uri>` = uniform resource identifier, `/path/to/file.ogg` is an example uri.

2.1.1 How to connect to the telnet server

The command `telnet localhost 1234` opens a telnet session on the local host at port 1234, the port liquidsoap is running on by default. You can also do `telnet server 1234` if liquidsoap is running on remote host `server`, and it's server is public – see settings for details.

Note: It is also possible to connect to liquidsoap remotely, even when it's not public: just tunnel the port via a ssh connection. Create the tunnel by running `ssh user@server -L 1234:localhost:1234`. Once the ssh connection has been established you can open another terminal and start the telnet using `telnet localhost 1234` as if you were on the server.

2.1.2 Scripting for the telnet server

Although it's possible to interact with liquidsoap manually via telnet, one will usually want to automate that interaction via scripts. For example, Perl and Ruby have simple telnet modules which allow it, as demonstrated in liquidsoap/scripts (http://savonet.svn.sourceforge.net/viewvc/savonet/trunk/liquidsoap/scripts/): `scripts/ask-liquidsoap.rb uptime`, `scripts/ask-liquidsoap.pl "request.push path/to/your/music file.ogg"`.

2.1.3 General commands

`help`

This lists the available commands.

```
help
Available commands:
| alive
| day.m3u.next
| exit
```

```
| help
| jingles.pls.next
| list
| metadata <rid>
| night.m3u.next
| on_air
| quit
| yourradio.ogg.metadatas
| yourradio.ogg.remaining
| yourradio.ogg.skip
| yourradio.ogg.start
| yourradio.ogg.status
| yourradio.ogg.stop
| request.consider <rid>
| request.ignore <rid>
| request.push <uri>
| request.queue
| resolving
| scheduler.consider <rid>
| scheduler.ignore <rid>
| scheduler.push <uri>
| scheduler.queue
| trace <rid>
| uptime
END
```

`uptime`

Displays liquidsoap's uptime.

```
uptime
0j 00h 27m 31s
END
```

`alive`

Show all alive requests (pending, resolving, or playing), that are in our example used by playlists and queues. This is displayed as a list of RIDs.

```
alive
7 6 5 4 3 2 1
END
```

`resolving`

Show all resolving requests, those which are being downloaded, synthesized, or whatever the protocol handler is doing to resolve them. The format is the same as `alive`.

`metadata <rid>`

Displays metadata for a request.

```
metadata 8
title="le silence"
temporary="false"
license="Licensed under http://creativecommons.org/licenses/by-nc-nd/2.0/fr/"
```

```
date="2004-10-01"
artist="...anabase*"
description="http://www.jamendo.com/ : Free music"
rid="8"
source_id="521"
tracknumber="4"
initial_uri="/path/to/your/files/04 - le silence.ogg"
source="request"
organization="http://www.jamendo.com/ : Free music"
status="ready"
filename="/path/to/your/files/04 - le silence.ogg"
2nd_queue_pos="0"
album="expdition vers l'intrieur"
comment="http://www.jamendo.com/ : Free music"
www="http://www.jamendo.com/?&184"
END
```

`trace <rid>`

Trace the history of a request's resolving. See concepts for details on resolving.

```
trace 10
[2006/11/12 16:46:20] Pushed ["/path/to/silence.ogg";...].
[2006/11/12 16:46:20] "/path/to/silence.ogg" entered the secondary queue : position #1
[2006/11/12 16:46:20] Entering the primary queue.
[2006/11/12 16:52:11] Currently on air.
END
```

`on_air`

Displays the requests that are currently being played. Most of the time there is only one, but in general there could be several, for example if you mix two playlists together.

```
on_air
8
END
```

`exit`

Exits the telnet connection.

```
exit
Connection closed by foreign host
```

`quit`

Just like `exit`.

2.1.4 Playlist commands

`<playlist>.next`

Displays the upcoming tracks for a playlist, for example `day.m3u`.

```
day.m3u.next

/path/to/your/files/04 - Le trottoir.ogg
/path/to/your/files/csr002-01-twizzle-falling.mp3
/path/to/your/files/GilbertoGil_Oslodum.ogg
/path/to/your/files/01 - Mme Asperge.mp3
/path/to/your/files/hb02_kosmo_knoedeltroete(rmx).mp3
/path/to/your/files/01_Colida_Carni-War.mp3
/path/to/your/files/01 - les trains.ogg
/path/to/your/files/04_DHS0015.mp3

END
```

2.1.5 Output commands

`<output>.metadatas`

Displays metadata of already or currently-being played tracks, a metadata history.

```
yourradio.ogg.metadatas
--- 10 ---
title="Passing"
temporary="false"
date="2006"
composer=
artist="Pablo Cepeda"
rid="5"
on_air="2006/11/12 16:36:11"
tracknumber="01"
initial_uri="/path/to/your/files/kpu088-pablo-cepeda-01-passing.mp3"
orig. artist=
source="day.m3u"
status="playing"
filename="/path/to/your/files/kpu088-pablo-cepeda-01-passing.mp3"
genre="(52)Electronic"
album="Le cicle du calme"
comment="http://www.kikapu.com"
--- 9 ---
etc="etc..."
END
```

`<output>.remaining`

Displays the remaining time in the current track, in seconds. This information is currently not very precise, and often simply unavailable, in which case the output is `(undef)`.

```
yourradio.ogg.remaining
7 sec
END
```

`<output>.skip`

Skips the currently playing track.

```
yourradio.ogg.skip
Done
END
```

`<output>.start`

Start streaming if paused.

```
yourradio.ogg.start

END
```

`<output>.stop`

Pauses streaming, keeps liquidsoap running. The current track is continued when the stream is restarted, unless it's been consumed by another output in the meantime.

```
yourradio.ogg.stop

END
```

`<output>.status` Displays the output's status. Obviously, if the stream is running it is "on", if it is paused it is "off".

```
yourradio.ogg.status
on
END
```

2.1.6 Queue commands

Queue sources (`request.queue` and `request.equeue`) have two queues. In the secondary one, requests are stored, waiting to be resolved. When needed, requests are taken out of the secondary queue, resolved, and then put into the primary one. Finally, one request is removed from the primary queue and played.

`<queue>.ignore <rid>`

Tells the queue to drop request `rid` when it is popped out of the secondary queue.

```
request.ignore 8
OK
END
```

`<queue>.consider <rid>`

Cancels a `ignore`, if it's not too late, i.e. if the request hasn't been popped and droped already. Same format as `ignore`.

`<queue>.push <uri>`

Pushes a request to the request queue.

```
request.push /path/to/your/files/04 - le silence.ogg
8
END
```

`<queue>.queue`

Displays the list of `<rid>` of the requests from both queues, and currently playing.

```
request.queue
8
END
```

2.2 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 broadcasted. 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 broadcasted; sometimes, the staff has so much fun (or is it something else ?) doing live shows that they live 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:

```
set log.dir = "/tmp"
set log.stdout = true

interlude = one_file("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(length=5.,mic()) , interlude ])

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

If you don't get the difference between these two operators, maybe you need to learn more about the basic concepts of Liquidsoap, especially the 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 whathever you want")
end
source = on_blank(handler,source)
```

2.3 Distributed encoding

Using RTP, liquidsoap can directly output the raw stream with metadata. Then you can set up another liquidsoap instance on an other machine of your network which just inputs this RTP

stream and encodes it, for example for sending to Icecast. It allows you to share the load on many machines, and also make the main liquidsoap process more independant of the Icecast servers. These can now crash or be restarted, you'll just have to restart the RTP encoders.

Here is how to setup a RTP server:

```
set log.dir = "/tmp"
set log.stdout = true
set server.port = 1235

output.rtp(single("/usr/share/mrpingouin/mp3bis/bodom/TheNail.ogg"))
```

And here is the client, takes the RTP stream and plays it on your speakers:

```
set log.dir = "/tmp"
set log.stdout = true

output.alsa(input.rtp())
```

The server port has been specified on the server to be different from the default 1234 used on the client, so that thet don't conflict if ran on the same host. If you want to run the client on another host, specify a sufficient TTL for the RTP output, default being 0: `output.rtp(ttl=1,...)`.

Chapter 3

Other tools

3.1 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 it 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)))
```

It allows us to 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 transform the query into a SQLite query and return a list of ten random matches, which liquidsoap will try.

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

3.2 Bottle

Bottle is a prototype IRC bot written in OCaml. It uses a modular plugin system and is in particular able to communicate with liquidsoap. Currently, it is able to:

- show information about the song currently playing,
- listen to users' song requests,
- skip songs.

It is an example of how to write software which interacts with liquidsoap. You can get its source code via SVN: `http://svn.sourceforge.net/savonet/trunk/bottle/`.

This kind of tools doesn't need to be done in OCaml. It is quite easy to write an interface module for liquidsoap in perl, python or ruby too. We have a perl module in an unpublished bot – available on request. A python module is available in liguidsoap.

Chapter 4

Reference

4.1 %

`(string, [(string*string)]) -> string`
(pattern % [...,(k,v),...]) replaces in pattern occurences of:
- '$(k)' into "v";
- '$(if $(k2),"a","b")' into "a" if k2 is found in the list, "b" otherwise.

4.2 ^

`(string, string) -> string`
Concatenate strings.

4.3 add

`(?id:string, ?normalize:bool, ?weights:[int], [source]) -> source`
Add sources, with normalization

- `(unlabeled)` (`[source]`)
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `normalize` (`bool` — defaults to `true`)
- `weights` (`[int]` — defaults to `[ ]`): Relative weight of the sources in the sum. The empty list stands for the homogeneous distribution.

4.4 add_protocol

`(string, ((string, float) -> [string])) -> unit`
Register a new protocol.

4.5 and

`(bool, bool) -> bool`
Return the conjunction of its arguments

4.6 append

`(?id:string, ?merge:bool, source, (([(string*string)]) -> source)) -> source`
Append an extra track to every track.Set the metadata 'liq_append' to 'false' to inhibit appending on one track.

- `(unlabeled)` (`([(string*string)]) -> source`): Given the metadata, build the source producing the track to append. This source is allowed to fail (produce nothing) if no relevant track is to be appended.
- `(unlabeled)` (`source`)
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `merge` (`bool` — defaults to `false`): Merge the track with its appended track.

4.7 assoc

`(string, [(string*string)]) -> string`
assoc k [...,(k,v),...] = v

4.8 blank

`(?id:string, ?duration:float) -> source`
This source is not very noisy :)

- `duration` (`float` — defaults to `0.`): Duration of blank tracks, default means forever.
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.9 change_volume

`(?id:string, float, source) -> source`
Scales the amplitude of the signal

- `(unlabeled)` (`source`)
- `(unlabeled)` (`float`): multiplicative factor
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.10 cross

`(?id:string, ?duration:float, ?inhibit:float, ?minimum:float, ((source, source) -> source), source) -> source`
Generic cross operator, allowing the composition of the N last seconds of a track with the beginning of the next track.

- `(unlabeled)` (`source`)
- `(unlabeled)` (`((source, source) -> source`): Composition of an end of track and the next track.
- `duration` (`float` — defaults to `5.`): Duration in seconds of the crossed end of track. This value can be set on a per-file basis using the metadata field 'liq_start_next' (float in seconds).

- `id` (`string` — defaults to `""`): Force the value of the source ID
- `inhibit` (`float` — defaults to `-1.`): Minimum delay between two transitions. It is useful in order to avoid that a transition is triggered on top of another when an end-of-track occurs in the first one. Negative values mean 'same as duration'.
- `minimum` (`float` — defaults to `-1.`): Minimum duration (in sec.) for a cross: If the track ends without any warning (e.g. in case of skip) there may not be enough data for a decent composition. Set to 0. to avoid having transitions after skips, or more to avoid transitions on short tracks. With the negative default, transitions always occur.

4.11 delay

`(?id:string, float, source) -> source`
Prevents the child from being ready again too fast after a end of track

- `(unlabeled)` (`source`)
- `(unlabeled)` (`float`): The source won't be ready less than this amount of seconds after any end of track
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.12 fade.final

`(?id:string, ?duration:float, source) -> source`
Fade a stream to silence.

- `(unlabeled)` (`source`)
- `duration` (`float` — defaults to `3.`)
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.13 fade.in

`(?id:string, ?duration:float, source) -> source`
Fade the beginning of tracks. Metadata 'liq_fade_in' can be used to set the duration for a specific track (float in seconds).

- `(unlabeled)` (`source`)
- `duration` (`float` — defaults to `3.`)
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.14 fade.initial

`(?id:string, ?duration:float, source) -> source`
Fade the beginning of a stream.

- `(unlabeled)` (`source`)
- `duration` (`float` — defaults to `3.`)
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.15 fade.out

`(?id:string, ?duration:float, source) -> source`
Fade the end of tracks. Metadata 'liq_fade_out' can be used to set the duration for a specific track (float in seconds).

- `(unlabeled)` (`source`)
- `duration` (`float` — defaults to `3.`)
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.16 fallback

`(?id:string, ?track_sensitive:bool, ?before:float, ?transitions:[(source, source) -> source], [source]) -> source`
At the beginning of each track, select the first ready child.

- `(unlabeled)` (`[source]`): Select the first ready source in this list.
- `before` (`float` — defaults to `0.`): EXPERIMENTAL: for track_sensitive switches, trigger transitions before the end of track.
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `track_sensitive` (`bool` — defaults to `true`): Re-select only on end of tracks
- `transitions` (`[(source, source) -> source]` — defaults to `[ ]`): Transition functions, padded with (fun (x,y) -¿ y) functions.

4.17 filter

`(?id:string, ~freq:int, ~q:float, ~mode:string, source) -> source`
Perform several kinds of filtering on the signal

- `(unlabeled)` (`source`)
- `freq` (`int`)
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `mode` (`string`): low—high—band—notch
- `q` (`float`)

4.18 get_process_lines

`(string) -> [string]`
Perform a shell call and return the list of its output lines.

4.19 get_process_output

`(string) -> string`
Perform a shell call and return its output.

4.20 input.http

`(?id:string, ?autostart:bool, string) -> source`
Forwards the given ogg/vorbis http stream. The relay can be paused/resumed using the start/stop telnet commands.

- `(unlabeled)` (`string`): URL of an http ogg stream (default port is 8000).
- `autostart` (`bool` — defaults to `true`): Initially start relaying or not.
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.21 input.http.mp3

`(?id:string, string, ?autostart:bool, ?samplefreq:int) -> source`
Forwards the given MP3 http stream. The relay can be paused/resumed using the start/stop telnet commands.

- `(unlabeled)` (`string`): URL of an http mp3 stream (default port is 8000).
- `autostart` (`bool` — defaults to `true`): Initially start relaying or not.
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `samplefreq` (`int` — defaults to `44100`): Samplefreq of the input stream.

4.22 mix

`(?id:string, [source]) -> source`
Mixing table controllable via the telnet interface.

- `(unlabeled)` (`[source]`)
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.23 on_blank

`(?id:string, (() -> unit), ?threshold:float, ?length:float, source) -> source`
Calls a given handler when detecting a blank

- `(unlabeled)` (`source`)
- `(unlabeled)` (`() -> unit`)
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `length` (`float` — defaults to `20.`): Maximum silence length allowed.
- `threshold` (`float` — defaults to `100.`): Intensity threshold under which the stream is considered to be blank.

4.24 on_metadata

`(?id:string, (([(string*string)]) -> unit), source) -> source`
Call a given handler on metadata packets.

- `(unlabeled)` (`source`)
- `(unlabeled)` (`([(string*string)]) -> unit`): Function called on every metadata packet in the stream. It should be fast because it is ran in the main thread.
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.25 or

`(bool, bool) -> bool`
Return the disjunction of its arguments

4.26 output.ao

`(?id:string, ?start:bool, ?driver:string, ?options:[(string*string)], source) -> source`
Output stream to local sound card using libao.

- `(unlabeled)` (`source`)
- `driver` (`string` — defaults to `""`): libao driver to use
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `options` (`[(string*string)]` — defaults to `[ ]`): List of parameters, depends on driver
- `start` (`bool` — defaults to `true`): Start output on operator initialization.

4.27 output.dummy

`(?id:string, source) -> source`
Dummy output for debugging purposes.

- `(unlabeled)` (`source`)
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.28 output.icecast

`(?id:string, ?start:bool, ?host:string, ?port:int, ?user:string, ?password:string, ?genre:string, ?url:string, ?description:string, ?public:bool, ?multicast_ip:string, ?mount:string, ?name:string, ?bitrate:int, ?quality:float, ?freq:int, ?stereo:bool, source) -> source`
Send a Vorbis stream to an icecast-compatible server.

- `(unlabeled)` (`source`)
- `bitrate` (`int` — defaults to `-1`)
- `description` (`string` — defaults to `"OCaml Radio!"`)

- `freq` (`int` — defaults to `44100`)
- `genre` (`string` — defaults to `"Misc"`)
- `host` (`string` — defaults to `"localhost"`)
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `mount` (`string` — defaults to `"Use [name].ogg"`)
- `multicast_ip` (`string` — defaults to `"no_multicast"`)
- `name` (`string` — defaults to `"Use [mount]"`)
- `password` (`string` — defaults to `"hackme"`)
- `port` (`int` — defaults to `8000`)
- `public` (`bool` — defaults to `true`)
- `quality` (`float` — defaults to `0.5`)
- `start` (`bool` — defaults to `true`): Start output threads on operator initialization.
- `stereo` (`bool` — defaults to `true`)
- `url` (`string` — defaults to `"http://savonet.sf.net"`)
- `user` (`string` — defaults to `"source"`)

4.29 output.ogg

`(?id:string, ?start:bool, ?quality:float, ?bitrate:int, ?freq:int, ?stereo:bool, string, source) -> source`

Output the source's stream as an OGG file.

- `(unlabeled)` (`source`)
- `(unlabeled)` (`string`): Filename where to output the OGG stream.
- `bitrate` (`int` — defaults to `-1`)
- `freq` (`int` — defaults to `44100`)
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `quality` (`float` — defaults to `0.5`)
- `start` (`bool` — defaults to `true`): Start output threads on operator initialization.
- `stereo` (`bool` — defaults to `true`)

4.30 output.wav

`(?id:string, ?start:bool, string, source) -> source`

Output the source's stream to a WAV file.

- `(unlabeled)` (`source`)
- `(unlabeled)` (`string`)
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `start` (`bool` — defaults to `true`)

4.31 pipe

`(?id:string, string, source) -> source`

Filter data through an external process. The process should read raw CD format on stdin and write the same on stdout. The command should output one frame for every input frame, and flush its output. Otherwise, liquidsoap will block.

- `(unlabeled)` (`source`)
- `(unlabeled)` (`string`)
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.32 playlist

`(?id:string, ?length:float, ?default_duration:float, ?timeout:float, ?mode:string, ?reload:int, ?reload_mode:string, string) -> source`

Loop on a playlist of URIs.

- `(unlabeled)` (`string`): URI where to find the playlist
- `default_duration` (`float` — defaults to `30.`): When unknown, assume this duration (in sec.) for files.
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `length` (`float` — defaults to `60.`): How much audio (in sec.) should be downloaded in advance.
- `mode` (`string` — defaults to `"randomize"`): normal—random—randomize
- `reload` (`int` — defaults to `0`): Amount of time (in seconds or rounds) before which the playlist is reloaded; 0 means never.
- `reload_mode` (`string` — defaults to `"seconds"`): rounds—seconds: unit of the 'reload' parameter
- `timeout` (`float` — defaults to `20.`): Timeout (in sec.) for a single download.

4.33 playlist.safe

`(?id:string, ?mode:string, ?reload:int, ?reload_mode:string, string) -> source`

Loop on a playlist of local files, and never fail. In order to do so, it has to check every file at the loading, so the streamer startup may take a few seconds. To avoid this, use a standard playlist, and put only a few local files in a default safe_playlist in order to ensure the liveness of the streamer.

- `(unlabeled)` (`string`): URI where to find the playlist
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `mode` (`string` — defaults to `"randomize"`): normal—random—randomize
- `reload` (`int` — defaults to `0`): Amount of time (in seconds or rounds) before which the playlist is reloaded; 0 means never.
- `reload_mode` (`string` — defaults to `"seconds"`): rounds—seconds: unit of the 'reload' parameter

4.34 prepend

`(?id:string, ?merge:bool, source, (([(string*string)]) -> source)) -> source`

Prepend an extra track before every track. Set the metadata 'liq_prepend' to 'false' to inhibit prepending on one track.

- `(unlabeled)` (`([(string*string)]) -> source`): Given the metadata, build the source producing the track to prepend. This source is allowed to fail (produce nothing) if no relevant track is to be appended. However, success must be immediate.
- `(unlabeled)` (`source`)
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `merge` (`bool` — defaults to `false`): Merge the track with its appended track.

4.35 quote

`(string) -> string`

Escape shell metacharacters.

4.36 random

`(?id:string, ?track_sensitive:bool, ?before:float, ?transitions:[(source, source) -> source], ?weights:[int], ?strict:bool, [source]) -> source`

At the beginning of every track, select a random ready child.

- `(unlabeled)` (`[source]`)
- `before` (`float` — defaults to `0.`): EXPERIMENTAL: for track_sensitive switches, trigger transitions before the end of track.
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `strict` (`bool` — defaults to `false`): Do not use random but cycle over the uniform distribution.
- `track_sensitive` (`bool` — defaults to `true`): Re-select only on end of tracks
- `transitions` (`[(source, source) -> source]` — defaults to `[ ]`): Transition functions, padded with (fun (x,y) -¿ y) functions.
- `weights` (`[int]` — defaults to `[ ]`): Weights of the children in the choice.

4.37 request

`(?indicators:[string], string) -> request`

Create a request.

4.38 request.dynamic

`(?id:string, (() -> request), ?length:float, ?default_duration:float, ?timeout:float) -> source`

Play request dynamically created by a given function.

- `(unlabeled)` (`() -> request`): A function generating requests: an initial URI (possibly fake) together with an initial list of alternative indicators
- `default_duration` (`float` — defaults to `30.`): When unknown, assume this duration (in sec.) for files.
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `length` (`float` — defaults to `60.`): How much audio (in sec.) should be downloaded in advance.
- `timeout` (`float` — defaults to `20.`): Timeout (in sec.) for a single download.

4.39 request.equeue

`(?id:string, ?length:float, ?default_duration:float, ?timeout:float) -> source`

Receive URIs from users, and play them. Insertion and deletion possible at any position.

- `default_duration` (`float` — defaults to `30.`): When unknown, assume this duration (in sec.) for files.
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `length` (`float` — defaults to `60.`): How much audio (in sec.) should be downloaded in advance.
- `timeout` (`float` — defaults to `20.`): Timeout (in sec.) for a single download.

4.40 request.queue

`(?id:string, ?length:float, ?default_duration:float, ?timeout:float) -> source`

Receive URIs from users, and play them.

- `default_duration` (`float` — defaults to `30.`): When unknown, assume this duration (in sec.) for files.
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `length` (`float` — defaults to `60.`): How much audio (in sec.) should be downloaded in advance.
- `timeout` (`float` — defaults to `20.`): Timeout (in sec.) for a single download.

4.41 rewrite_metadata

`(?id:string, [(string*string)], source) -> source`

Rewrite metadata on the fly.

- `(unlabeled)` (`source`)
- `(unlabeled)` (`[(string*string)]`): List of (target,value) rewriting rules.
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.42 sequence

`(?id:string, ?merge:bool, [source]) -> source`
Play only one track of every successive source, except for the last one which is played as much as available.

- `(unlabeled)` (`[source]`)
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `merge` (`bool` — defaults to `false`)

4.43 sine

`(?id:string, ?duration:float, int) -> source`
Plays a boring sine...

- `(unlabeled)` (`int`): Frequency of the sine
- `duration` (`float` — defaults to `0.`)
- `id` (`string` — defaults to `""`): Force the value of the source ID

4.44 single

`(?id:string, string, ?length:float, ?default_duration:float, ?timeout:float) -> source`
Loop on a request. It never fails if the request is static, meaning that it can be fetched once. Typically, http, ftp, say requests are static, and time is not.

- `(unlabeled)` (`string`): URI where to find the file
- `default_duration` (`float` — defaults to `30.`): When unknown, assume this duration (in sec.) for files.
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `length` (`float` — defaults to `60.`): How much audio (in sec.) should be downloaded in advance.
- `timeout` (`float` — defaults to `20.`): Timeout (in sec.) for a single download.

4.45 skip_blank

`(?id:string, ?threshold:float, ?length:float, source) -> source`
Skip track when detecting a blank

- `(unlabeled)` (`source`)
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `length` (`float` — defaults to `20.`): Maximum silence length allowed.
- `threshold` (`float` — defaults to `100.`): Intensity threshold under which the stream is considered to be blank.

4.46 store_metadata

`(?id:string, ?size:int, source) -> source`
Keep track of the last N metadata packets in the stream, and make the history available via a server command.

- `(unlabeled)` (`source`)
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `size` (`int` — defaults to `10`): Size of the history

4.47 strip_blank

`(?id:string, ?threshold:float, ?length:float, source) -> source`
Strip blanks

- `(unlabeled)` (`source`)
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `length` (`float` — defaults to `20.`): Maximum silence length allowed.
- `threshold` (`float` — defaults to `100.`): Intensity threshold under which the stream is considered to be blank.

4.48 switch

`(?id:string, ?track_sensitive:bool, ?before:float, ?transitions:[(source, source) -> source], ?strict:bool, ?single:[bool], [((() -> bool)*source)]) -> source`
At the beginning of a track, select the first source Si such than the temporal predicate Ii is true.

- `(unlabeled)` (`[((() -> bool)*source)]`): Sources Si with the interval Ii when they should be played.
- `before` (`float` — defaults to `0.`): EXPERIMENTAL: for track_sensitive switches, trigger transitions before the end of track.
- `id` (`string` — defaults to `""`): Force the value of the source ID
- `single` (`[bool]` — defaults to `[ ]`): Forbid the selection of a branch for two tracks in a row. The empty list stands for [false,...,false].
- `strict` (`bool` — defaults to `false`): Unset the operator's ready flag as soon as there is no valid interval, possibly interrupting ongoing tracks.
- `track_sensitive` (`bool` — defaults to `true`): Re-select only on end of tracks
- `transitions` (`[(source, source) -> source]` — defaults to `[ ]`): Transition functions, padded with (fun (x,y) -¿ y) functions.

4.49 system

`(string) -> unit`
Shell command call.

4.50 time_in_mod

`(int, int, int) -> bool`

INTERNAL: time_in_mod(a,b,c) checks that the unix time T satisfies a ¡= T mod c ¡ b