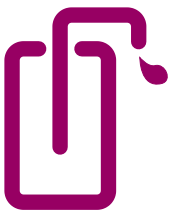
The Liquidsoap book

Samuel Mimram Romain Beauxis



Contents

1 Prologue

1.1 What is Liquidsoap?

So, you want to make a webradio? At first, this looks like an easy task, we simply need a program which takes a playlist of MP3 files, and broadcasts them one by one over the internet. However, in practice, people want something much more elaborate than just this.

For instance, we want to be able to switch between multiple playlists depending on the current time, so that we can have different ambiances during the day (soft music in the morning and techno at night). We also want to be able to incorporate requests from users (for instance, they could ask for specific songs on the website of the radio, or we could have guest DJ shows). Moreover, the music does not necessarily come from files which are stored on a local harddisk: we should be able to relay other audio streams, YouTube videos, or a microphone which is being recorded on the soundcard or on a distant computer.

When we start from music files, we rarely simply play one after the other. Generally, we want to have some fading between songs, so that the transition from a track to the next one is not too abrupt, and serious people want to be able to specify the precise time at which this fading should be performed on a per-song basis. We also want to add jingles between songs so that people know and remember about our radio, and to use speech synthesis to give the name of the song we just played. We should also maybe add commercials, so that we can earn some money, and those should be played at a given fixed hour, but should wait for the current song to be finished before launching the ad.

Also, we want to have some signal processing on our music in order to have a nice and even sound. We should adjust the gain so that successive tracks roughly have the same volume. We should also compress and equalize the sound in order to have a warm sound and to give the radio a unique color.

Finally, the rule number one of a webradio is that *it should never fail.* We want to ensure that if, for some reason, the stream we are usually relaying is not available, or the external harddisk on which our MP3 files are stored is disconnected, an emergency playlist will be played, so that we do not simply kick off our beloved listeners. More difficult, if the microphone of the speaker is unplugged, the soundcard will not be aware of it and will provide us with silence: we should be able to detect that we are streaming blank and after some time also fallback to the emergency playlist.

Once we have successfully generated the stream we had in mind, we need to encode it in order to have data which is small enough to be sent in realtime through the network. We actually often want to perform multiple simultaneous encodings of the same stream: this is necessary to account for various qualities (so that users can choose depending on their bandwidth) and various formats. We should also be able to broadcast those encoded streams using the various usual protocols that everybody uses nowadays.

As we can see, there is a wide variety of usages and technologies, and this is only the tip of the iceberg. Even more complex setups can be looked for in practice, especially if we have some form of interaction (through a website, a chatbot, etc.). Most other software tools to generate webradios impose a fixed workflow for webradios: they either consist in a graphical interface, which generally offers the possibility of programming a grid with different broadcasts on different timeslots, or a commandline program with more or less complex configuration files. As soon as your setup does not fit within the predetermined radio workflow, you are in trouble.

A new programming language. Based on this observation, we decided to come up with a new *programming language*, our beloved *Liquidsoap*, which would allow for describing how to generate audio streams. The generation of the stream does not need to follow a particular pattern here, it is instead implemented by the user in a script, which combines the various building blocks of the language in an arbitrary way: the possibilities are thus virtually unlimited. It does not impose a rigid approach for designing your radio, it can cope with all the situations described above, and much more.

Liquidsoap itself is programmed in the OCaml programming language, but the language you will use is not OCaml (although it was somewhat inspired of it), it is a new language, and it is quite different from a general-purpose programming language, such as Java or C. It was designed from scratch, with stream generation in mind, trying to follow the principle formulated by Allan Kay: *simple things should be simple, complex things should be possible*. This means that we had in mind that our users are not typically experienced programmers, but rather people enthusiastic about music or willing to disseminate information, and we wanted a language as accessible as possible, were a basic script should be easy to write and simple to understand, where the functions have reasonable default values, where the errors are clearly located and explained. Yet, we provide most things needed for handling sound (in particular, support for the wide variety of file formats, protocols, sound plugins, and so on) as well as more advanced functions which ensure that one can cope up with complex setups (e.g. through callbacks and references).

It is also designed to be very robust, since we want our radios to stream forever and not have our stream crash after a few weeks because of a rare case which is badly handled. For this reason, before running a script, the Liquidsoap compiler performs many in-depth checks on it, in order to ensure that everything will go on well. Most of this analysis is performed using *typing*, which offer very strong guarantees.

- We ensure that the data passed to function is of the expected form. For instance, the user cannot pass a string to a function expecting an integer. Believe it or not, this simple kind of error is the source of an incredible number of bugs in everyday programs (ask Python or Javascript programmers).
- We ensure there is always something to stream: if there is a slight chance that a source of sound might fail (e.g. the server on which the files are stored gets disconnected), Liquidsoap imposes that there should be a fallback source of sound.
- We ensure that we correctly handle synchronization issues. Two sources of sound (such as two soundcards) generally produce the sound at slightly different rates (the

promised 44100 samples per seconds might actually be 44100.003 for one and 44099.99 for the other). While slightly imprecise timing cannot be heard, the difference between the two sources accumulates on the long run and can lead to blanks (or worse) in the resulting sound. Liquidsoap imposes that a script will be able to cope with such situations (typically by using buffers).

Again, these potential errors are not detected while running the script, but before, and the experience shows that this results in quite robust sound production. In this book, we will mainly focus on applications and will not detail much further the theory behind those features of the language. If you want to know more about it, you can have a look at the two articles published on the subject, which are referenced at the end of the book (Baelde and Mimram 2008; Baelde, Beauxis, and Mimram 2011).

While we are insisting on webradios because this is the original purpose of Liquidsoap, the language is now also able to handle video. In some sense, this is quite logical since the production of a video stream is quite similar to the one of an audio stream, if we abstract away from technical details. Moreover, many radios are also streaming on YouTube, adding an image or a video, and maybe some information text sliding at the bottom.

Free software. The Liquidsoap language is a *free software*. This of course means that it is available for free on the internet, see the installation chapter, and more: this also means that the source code of Liquidsoap is available for you to study, modify and redistribute. Thus, you are not doomed if a specific feature is missing in the language: you can add it if you have the competences for that, or hire someone who does. This is guaranteed by the license of Liquidsoap, which is the *GNU General Public Licence 2* (and most of the libraries used by Liquidsoap are under the *GNU Lesser General Public Licence 2.1*).

Liquidsoap will always be free, but this does not prevent companies from selling products based on the language (and there are quite a number of those, be they graphical interfaces, web interfaces, or providing webradio tools as part of larger journalism tools) or services around the language (such as consulting). The main constraint imposed by the license is that if you distribute a modified version of Liquidsoap, say with some new features, you have to provide the user with the source code containing your improvements, under the same license as the original code.

The above does not apply to the current text which is covered by standard copyright laws.

A bit of history. The Liquidsoap language was initiated by David Baelde and Samuel Mimram, while students at the École Normale Supérieure de Lyon, around 2004. They liked to listen to music while coding and it was fun to listen to music together. This motivated David to write a Perl script based on the IceS program in order to stream their own radio on the campus: *geekradio* was born.

They did not have that many music files, and at that time it was not so easy to get online streams. But there were plenty of mp3s available on the internal network of the campus, which were shared by the students. In order to have access to those more easily, Samuel wrote a dirty campus indexer in OCaml (called *strider*, later on replaced by *bubble*), 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 started to build the first audio streamer in pure OCaml, using libshout. It had a simple telnet interface, so that a bot on IRC (this was the chat at that time) could send user requests easily to it, as well as from 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 some scheduling, especially techno music at night to code better.

Around that time, students had to propose and realize a "large" software project for one of their courses, with the whole class of around 30 students. David and Samuel proposed to build a complete flexible webradio system called *savonet* (an acronym of something like *Samuel and David's OCaml network*). A complete rewriting of every part of the streamer in OCaml was planned, with grand goals, so that everybody would have something to do: a new website with so many features, a new intelligent multilingual bot, new network libraries for glueing that, etc. Most of those did not survive up to now. But still, *Liquidsoap* was born, and plenty of new libraries for handling sound in OCaml emerged, many of which we are still using today. The name of the language was a play on word around "savonet" which sounds like "savonette", a soap bar in French.

On the day when the project had to be presented to the teachers, the demo miserably failed, but soon after that they were able to run a webradio with several static (but periodically reloaded) playlists, scheduled on different times, with a jingle added to the usual stream every hour, with the possibility of live interventions, allowing for user requests via a bot on IRC which would find songs on the database of the campus, which have priority over static playlists but not live shows, and added speech-synthetized metadata information at the end of requests.

Later on, the two main developers were joined by Romain Beauxis who was doing his PhD at the same place as David, and was also a radio enthusiastic: he was part of *Radio Pi*, the radio of École Centrale in Paris, which was soon entirely revamped and enhanced thanks to Liquidsoap. Over the recent year, he has become the main maintainer (taking care of the releases) and developer of Liquidsoap (adding, among other, support for FFmpeg in the language).

1.2 About this book

Prerequisites. We expect that the computer knowledge can vary much between Liquidsoap users, who can range from music enthusiasts to experienced programmers, and we tried to accommodate with all those backgrounds. Nevertheless, we have to suppose that the reader of this book is familiar with some basic concepts and tools. In particular, this book does not cover the basics of text file editing and Unix shell scripting (how to use the command line, how to run a program, and so on). Some knowledge in signal processing, streaming and programming can also be useful.

Liquidsoap version. The language has gone through some major changes since its beginning and maintaining full backward-compatibility was impossible. In this book, we assume that you have a version of Liquidsoap which is at least 2.0. Most examples could easily be adapted to work with earlier versions though, at the cost of making minor changes.

How to read the book. This book is intended to be read mostly sequentially, excepting perhaps this chapter, where we present the whole language in details, which can be skimmed trough at first. It is meant as a way of learning Liquidsoap, not as a 500+ pages references manual: should you need details about the arguments of a particular operator, you are advised to have a look at the online documentation.

We explain the technological challenges that we have face in order to produce multimedia streams in this chapter and are addressed by Liquidsoap. The means of installing the software

are described in this chapter. We then describe in this chapter what everybody wants to start with: setting up a simple webradio station. Before, going to more advanced uses, we first need to understand what we can do in this language, and this is the purpose of this chapter. We then detail the various ways to generate a webradio in there and a video stream in there. Finally, for interested readers, we give details about the internals of the language and the production of streams in there.

How to get help. You are reading the book and still have questions? There are many ways to get in touch with the community and obtain help to get your problem solved:

1. the Liquidsoap website contains an extensive up-to-date documentation and tutorials about specific points,
2. the Liquidsoap slack workspace is a public chat on where you can have instantaneous discussions,
3. the Liquidsoap mailing-list is there if you would rather discuss by mail (how old are you?),
4. the Liquidsoap github page is the place to report bugs,
5. there is also a Liquidsoap discussion board.

Please remember to be kind, most of the people there are doing this on their free time!

How to improve the book. We did our best to provide a progressive and illustrated introduction to Liquidsoap, which covers almost all the language, including the most advanced features. However, we are open to suggestions: if you find some error, some unclear explanation, or some missing topic, please tell us! The best way is by opening an issue on the dedicated bugtracker, but you can also reach us by mail at `sam@liquidsoap.info` and `romain@liquidsoap.info`. Please include page numbers and text excerpts if your comment applies to a particular point of the book (or, better, make pull requests). The version you are holding in your hands was compiled on November 23, 2021, you can expect frequent revisions to fix found issues.

The authors. The authors of the book you have in your hands are the two main current developers of Liquidsoap. *Samuel Mimram* obtained his PhD in computer science 2009 and is currently a Professor in computer science in École polytechnique, France. *Romain Beauxis* obtained his PhD in computer science in 2009 and is currently a software engineer based in New Orleans.

Thanks. The advent of Liquidsoap and this book would not have been possible without the numerous contributors over the years, the first of them being David Baelde who was a leading creator and designer of the language, but also the students of the MIM1 (big up to Florent Bouchez, Julien Cristau, Stéphane Gimenez and Sattisvar Tandabany), and our fellow users Gilles Pietri, Clément Renard and Vincent Tabard (who also designed the logo), as well as all the regulars of slack and the mailing-list. Many thanks also to the many people who helped to improve the language by reporting bugs or suggesting ideas, and to the Radio France team who where enthusiastic about the project and motivated some new developments (hello Maxime Bugeia, Youenn Piolet and others).

2 The technology behind streams

Before getting our hands on Liquidsoap, let us quickly describe the typical toolchain involved in a webradio, in case the reader is not familiar with it. It typically consists of the following three elements.

The *stream generator* is a program which generates an audio stream, generally in compressed form such as MP3 or AAC, be it from playlists, live sources, and so on. Liquidsoap is one of those and we will be most exclusively concerned with it, but there are other friendly competitors ranging from Ezstream, IceS or DarkIce which are simple command-line free software that can stream a live input or a playlist to an Icecast server, to Rivendell or SAM Broadcaster which are graphical interfaces to handle the scheduling of your radio. Nowadays, websites are also proposing to do this online on the cloud; these include AzuraCast, Centova and Radionomy which are all powered by Liquidsoap!

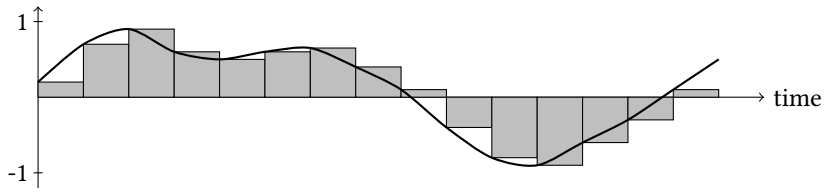
A *streaming media system*, which is generally Icecast. Its role is to relay the stream from the generator to the listeners, of which there can be thousands. With the advent of HLS, it tends to be more and more replaced by a traditional web server.

A *media player*, which connects to the server and plays the stream for the client, it can either be a software (such as iTunes), an Android application, or a website.

Since we are mostly concerned with stream generation, we shall begin by describing the main technological challenges behind it.

2.1 Audio streams

Digital audio. Sound consists in regular vibrations of the ambient air, going back and forth, which you perceive through the displacements of the tympanic membrane that they induce in your ear. In order to be represented in a computer, such a sound is usually captured by a microphone, which also has a membrane, and is represented by samples, corresponding to the successive positions of the membrane of the microphone. In general, sound is sampled at 44.1 kHz, which means that samples are captured 44100 times per second, and indicate the position of the membrane, which is represented by a floating point number, conventionally between -1 and 1. In the figure below, the position of the membrane is represented by the continuous black curve and the successive samples correspond to the grayed rectangles:



The way this data is represented is a matter of convention and many of those can be found in "nature":

- the sampling rate is typically 44.1 kHz (this is for instance the case in audio CDs), but the movie industry likes more 48 kHz, and recent equipment and studios use higher rates for better precision (e.g. DVDs are sampled at 92 kHz),
- the representation of samples varies: Liquidsoap internally uses floats between -1 and 1 (stored in double precision with 64 bits), but other conventions exist (e.g. CDs use 16 bits integers ranging from -32768 to 32767, and 24 bits integers are also common).

In any case, this means lots of data. For instance, an audio sample in CD quality takes 2 bytes (= 16 bits, remember that a byte is 8 bits) for each of the 2 channels and 1 minute of sound is 44100×2×2×60 bytes, which is roughly 10 MB per minute.

Compression. Because of the large quantities of data involved, sound is typically compressed, especially if you want to send it over the internet where the bandwidth, i.e. the quantity of information you can send in a given period of time, matters: it is not unlimited and it costs money. To give you an idea, a typical fiber connection nowadays has an upload rate of 100 megabits per second, with which you can send CD quality audio to roughly 70 listeners only.

One way to compress audio consists in using the standard tools from coding and information theory: if something occurs often then encode it with a small sequence of bytes (this is how compression formats such as zip work for instance). The *FLAC* format uses this principle and generally achieves compression to around 65% of the original size. This compression format is *lossless*, which means that if you compress and then decompress an audio file, you will get back to the exact same file you started with.

In order to achieve more compression, we should be prepared to lose some data present in the original file. Most compressed audio formats are based, in addition to the previous ideas, on psychoacoustic models which take in account the way sound is perceived by the human hear and processed by the human brain. For instance, the ears are much more sensitive in the 1 to 5 kHz range so that we can be more rough outside this range, some low intensity signals can be masked by high intensity signals (i.e., we do not hear them anymore in presence of other loud sound sources), they do not generally perceive phase difference under a certain frequency so that all audio data below that threshold can be encoded in mono, and so on. Most compression formats are *destructive*: they remove some information in the original signal in order for it to be smaller. The most well-known are MP3, Opus and AAC: the one you want to use is a matter of taste and support on the user-end. The MP3 format is the most widespread, the Opus format has the advantage of being open-source and patent-free, has a good quality/bandwidth radio and is reasonably supported by modern browsers but hardware support is almost nonexistent, the AAC format is proprietary so that good free encoders are more difficult to find (because they are subject to licensing fees) but achieves good sounding at high compression rates and is quite well supported, etc. A typical MP3 is encoded at a bitrate of 128 kbps (kilobits per second, although rates of 192 kbps and higher

are recommended if you favor sound quality), meaning that 1 minute will weight roughly 1 MB, which is 10% of the original sound in CD quality.

Most of these formats also support *variable bitrates* meaning that the bitrate can be adapted within the file: complex parts of the audio will be encoded at higher rates and simpler ones at low rates. For those, the resulting stream size will heavily depend on the actual audio and is thus more difficult to predict, by the perceived quality is higher.

As a side note, we were a bit imprecise above when speaking of a "file format" and we should distinguish between two things: the *codec* which is the algorithm we used to compress the audio data, and the *container* which is the file format used to store the compressed data. This is why one generally speaks of ogg/opus: Ogg is the container and Opus is the codec. A container can usually embed streams encoded with various codecs (e.g. ogg can also contain FLAC or vorbis streams), and a given codec can be embedded in various containers (e.g. flac and vorbis streams can also be embedded into Matroska containers). In particular, for video streams, the container typically contains multiple streams (one for video and one for audio), each encoded with a different codec, as well as other information (metadata, subtitles, etc.).

Metadata. Most audio streams are equipped with *metadata* which are textual information describing the contents of the audio. A typical music file will contain, as metadata, the title, the artist, the album name, the year of recording, and so on. Custom metadata are also useful to indicate the loudness of the file, the desired cue points, and so on.

2.2 Streaming

Once properly encoded, the streaming of audio data is generally not performed directly by the stream generator (such as Liquidsoap) to the client, a streaming server generally takes care of this. One reason to want separate tools is for reliability: if the streaming server gets down at some point because too many auditors connect simultaneously at some point, we still want the stream generator to work so that the physical radio or the archiving are still operational.

Another reason is that this is a quite technical task. In order to be transported, the streams have to be split in small packets in such a way that a listener can easily start listening to a stream in the middle and can bear the loss of some of them. Moreover, the time the data takes from the server to the client can vary over time (depending on the load of the network or the route taken): in order to cope with this, the clients do not play the received data immediately, but store some of it in advance, so that they still have something to play if next part of the stream comes late, this is called *buffering*. Finally, one machine is never enough to face the whole internet, so we should have the possibility of distributing the workload over multiple servers in order to handle large amounts of simultaneous connections.

Icecast. Historically, Icecast was the main open-source server used in order to serve streams over the internet. On a first connection, the client starts by buffering audio (in order to be able to cope with possible slowdowns of the network): Icecast therefore begins by feeding it up as fast as possible and then sends the data at a peaceful rate. It also takes care of handling multiple stream generators (which are called *mountpoints* in its terminology), multiple clients, replaying metadata (so that we have the title of the current song even if we started listening to it in the middle), recording statistics, enforcing limits (on clients or bandwidth), and so on. Icecast servers support relaying streams from other servers, which is useful in order to distribute listening clients across multiple physical machines, when many of them are expected to connect simultaneously.

HLS. Until recently, the streaming model as offered by Icecast was predominant, but it suffers from two main drawbacks. Firstly, the connection has to be kept between the client and the server for the whole duration of the stream, which cannot be guaranteed in mobile contexts: when you connect with your smartphone, you frequently change networks or switch between wifi and 4G and the connection cannot be held during such events. In this case, the client has to make a new connection to the Icecast server, which in practice induces blanks and glitches in the audio for the listener. Another issue is that the data cannot be cached as it is done for web traffic, where it helps to lower latencies and bandwidth-related costs, because each connection can induce a different response.

For these reasons, new standards such as HLS (for *HTTP Live Stream*) or DASH (for *Dynamic Adaptive Streaming over HTTP*) have emerged, where the stream is provided as a rolling playlist of small files called segments: a playlist typically contains the last minute of audio split into segments of 2 seconds. Moreover, the playlist can indicate multiple versions of the stream with various formats and encoding qualities, so that the client can switch to a lower bitrate if the connection becomes bad, and back to higher bitrates when it is better again, without interrupting the stream: this is called *adaptative* streaming. Here, the files are downloaded one by one, and are served by a usual HTTP server. This means that we can reuse all the technology developed for those to scale up and improve the speed, such as load balancing and caching techniques typically provided by content delivery networks. It seems that such formats will take over audio distribution in the near future, and Liquidsoap already has support for them. Their only drawback is that they are more recent and thus less well supported on old clients, although this tends to be less and less the case.

RTMP. Finally, we would like to mention that, nowadays, streaming is more and more being delegated to big online platforms, such as YouTube or Twitch, because of their ease of use, both in terms of setup and of user experience. Those generally use another protocol, called RTMP (*Real-Time Messaging Protocol*), which is more suited to transmitting live streams, where it is more important to keep the latency low (i.e. transmit the information as close as possible to the instant where it happened) than keep its integrity (dropping small parts of the audio or video is considered acceptable).

2.3 Audio sources

In order to make a radio, one has to start with a primary source of audio. We give examples of such below.

Audio files. A typical radio starts with one or more *playlists*, which are lists of audio files. These can be stored in various places: they can either be on a local hard drive or on some distant server, and are identified using a URI (for *Uniform Resource Identifier*) which can be a path to a local file or something of the form `http://some/server/file.mp3` which indicates that the file should be accessed using the HTTP protocol (some other protocols should also be supported). There is a slight difference between local and distant files: in the case of local files, we have pretty good confidence that they will always be available (or at least we can check that this is the case), whereas for distant files the server might be unavailable, or just very slow, so that we have to take care of downloading the file in advance enough and be prepared to have fallback option in case the file is not ready in time. Finally, audio files can be in various formats (as described in the above section) and have to be decoded, which is why Liquidsoap depends on many libraries, in order to support as many formats as possible.

Even in the case of local files, the playlist might be *dynamic*: instead of knowing in advance

the list of all the files, the playlist can consist of a queue of *requests* made by users (e.g., via a website or a chatbot); we can even call a script which will return the next song to be played, depending on whichever parameters (for instance taking in account votes on a website).

Live inputs. A radio often features live shows. As in the old days, the speaker can be in the same room as in the server, in which case the sound is directly captured by a soundcard. But nowadays, live shows are made more and more from home, where the speaker will stream its voice to the radio by himself, and the radio will interrupt its contents and relay the stream. More generally, a radio should be able to relay other streams along with their metadata (e.g. when a program is shared between multiple radios) or other sources (e.g. a live YouTube channel).

As for distant files, we should be able to cleanly handle failures due to network. Another issue specific to live streams (as opposed to playlists) is that we do not have control over time: this is an issue for operations such as crossfading (see below) which requires shifting time and thus cannot be performed on realtime sources.

Synchronization. In order to provide samples at a regular pace, a source of sound has an *internal clock* which will tick regularly: each soundcard has a clock, your computer has a clock, the live streams are generated by things which have clocks. Now, suppose that you have two soundcards generating sound at 44100 Hz, meaning that their internal clocks both tick at 44100 Hz. Those are not infinitely precise and it might be the case that there is a slight difference if 1 Hz between the two (maybe one is ticking at 44099.6 Hz and the other one at 44100.6 Hz in reality). Usually, this is not a problem, but on the long run it will become one: this 1 Hz difference means that, after a day, one will be 2 seconds in advance compared to the other. For a radio which is supposed to be running for months, this will be an issue and the stream generator has to take care of that, typically by using buffers. This is not a theoretical issue: first versions of Liquidsoap did not carefully handle this and we did experience quite a few problems related to it.

2.4 Audio processing

Resampling. As explained in the above section, various files have various sampling rates. For instance, suppose that your radio is streaming at 48 kHz and that you want to play a file at 44.1 kHz. You will have to *resample* your file, i.e. change its sampling rate, which, in the present case, means that you will have to come up with new samples. There are various simple strategies for this such as copying the sample closest to a missing one, or doing a linear interpolation between the two closest. This is what Liquidsoap is doing if you don't have the right libraries enabled and, believe it or not (or better try it!), it sounds quite bad. Resampling is a complicated task to get right, and can be costly in terms of CPU if you want to achieve good quality. Whenever possible Liquidsoap uses `libsamplerate` library to achieve this task, which provides much better results than the naive implementation.

Normalization. The next thing you want to do is to *normalize* the sound, meaning that you want to adjust its volume in order to have roughly the same audio loudness between tracks: if they come from different sources (such as two different albums by two different artists) this is generally not the case.

A strategy to fix that is to use *automatic gain control*: the program can regularly measure the current audio loudness based, say, on the previous second of sound, and increase or decrease the volume depending on the value of the current level compared to the target one. This

has the advantage of being easy to set up and providing a homogeneous sound. However, while it is quite efficient when having voice over the radio, it is quite unnatural for music: if a song has a quiet introduction for instance, its volume will be pushed up and the song as a whole will not sound as usual.

Another strategy for music consists in pre-computing the loudness of each file. It can be performed each time a song is about to be played, but it is much more efficient to compute this in advance and store it as a metadata: the stream generator can then adjust the volume on a per-song basis based on this information. The standard for this way of proceeding is *ReplayGain* and there are a few efficient tools to achieve this task. It is also more natural than basic gain control, because it takes in account the way our ears perceive sound in order to compute loudness.

At this point, we should also indicate that there is a subtlety in the way we measure volume (and loudness). It can either be measured *linearly*, i.e. we indicate the amplification coefficient by which we should multiply the sound, or in *decibels*. The reason for having the two is that the first is more mathematically pleasant, whereas the second is closer to the way we perceive the sound. The relationship between linear l and decibel d measurements is not easy, the formulas relating the two are $d=20\log_{10}(l)$ and $l=10^{d/20}$. If your math classes are too far away, you should only remember that 0 dB means no amplification (we multiply by the amplification coefficient 1), adding 6 dB corresponds to multiplying by 2, and removing 6 dB corresponds to dividing by 2:

decibels	-12	-6	0	6	12	18
amplification	0.25	0.5	1	2	4	8

This means that an amplification of -12 dB corresponds to multiplying all the samples of the sound by 0.25, which amounts to dividing them by 4.

Transitions between songs. In order to ease the transition between songs, one generally uses *crossfading*, which consists in fading out one song (progressively lowering its volume to 0) while fading in the next one (progressively increasing its volume from 0). A simple approach can be to crossfade for, say, 3 seconds between the end of a song and a beginning of the next one, but serious people want to be able to choose the length and type of fading to apply depending on the song. And they also want to have *cue points*, which are metadata indicating where to start a song and where to end it: a long intro of a song might not be suitable for radio broadcasting and we might want to skip it. Another common practice when performing transitions between the tracks consists in adding *jingles*: those are short audio tracks generally saying the name of the radio or of the current show. In any way, people avoid simply playing one track after another (unless it is an album) because it sounds awkward to the listener: it does not feel like a proper radio, but rather like a simple playlist.

Equalization. The final thing you want to do is to give your radio an appreciable and recognizable sound. This can be achieved by applying a series of sound effects such as

- a *compressor* which gives a more uniform sound by amplifying quiet sounds,
- an *equalizer* which gives a signature to your radio by amplifying differently different frequency ranges (typically, simplifying a bit, you want to insist on bass if you play mostly lounge music in order to have a warm sound, or on treble if you have voices in order for them to be easy to understand),
- a *limiter* which lowers the sound when there are high-intensity peaks (we want to avoid clipping),

- a *gate* which reduce very low level sound in order for silence to be really silence and not low noise (in particular if you capture a microphone),
- and so on.

These descriptions are very rough and we urge the reader not accustomed to those basic components of signal processing to learn more about them. You will need those at some point of you want to make a professional sounding webradio.

The processing loop. Because we generally want to perform all those operations on audio signals, the typical processing loop will consist in

1. decoding audio files,
2. processing the audio (fading, equalizing, etc.),
3. encoding the audio,
4. streaming encoded audio.

If for some reason we do not want to perform any audio processing (for instance, if this processing was done offline, or if we are relaying some already processed audio stream) and if the encoding format is the same as the source format, there is no need to decode and then reencode the sound: we can directly stream the original encoded files. By default, Liquidsoap will always reencode files but this can be avoided if we want, see there.

2.5 Interaction

What we have described so far is more or less the direct adaptation of traditional radio techniques to the digital world. But with new tools come new usages, and a typical webradio generally requires more than the above features. In particular, we should be able to interact with other programs and services.

Interacting with other programs. Whichever tool you are going to use in order to generate your webradio, it is never going to support all the features that a user will require. At some point, the use of an obscure hardware interface, a particular database, or a specific web framework will be required by a client, which will not be supported out of the shelf by the tool. Or maybe you simply want to be able to reuse parts of the scripts that you spent years to write in your favorite language.

For this reason, a stream generator should be able to interact with other tools, by calling external programs or scripts, written in whichever language. For instance, we should be able to handle *dynamic playlists*, which are playlists where the list of songs is not determined in advance, but rather generated on the fly: each time a song ends a function of the generator or an external program computes the next song to be played.

We should also be able to easily import data generated by other programs, the usual mechanism being by reading the standard plain text output of the executed program. This means that we should also have tools to parse and manipulate this standard output. Typically, structured data such as the result of a query on a database can be output in standard formats such as JSON, for which we should have support.

Finally, we should be able to interact with some more specific external programs, such as for monitoring scripts (in order to understand its state and be quickly notified in case of a problem).

Interacting with other services. The above way of interacting works in *pull mode*: the stream generators asks an external program for information, such as the next song to be played. Another desirable workflow is in *push mode*, where the program adds information whenever it feels like. This is typically the case for *request queues* which are a variant of playlists, where an external programs can add songs whenever it feels like: those will be played one, in the order where they were inserted. This is typically used for interactive websites: whenever a user asks for a song, it gets added to the request queue.

Push mode interaction is also commonly used for controllers, which are physical or virtual devices consisting of push buttons and sliders, that one can use in order to switch between audio sources, change the volume of a stream, and so on. The device generally notifies the stream generator when some control gets changed, which should then react accordingly. The commonly used standard nowadays for communicating with controllers is called OSC (*Open Sound Control*).

2.6 Video streams

The workflow for generating video streams is not fundamentally different from the one that we have described above, so that it is natural to expect that an audio stream generator can also be used to generate video streams. In practice, this is rarely the case, because manipulating video is an order of magnitude harder to implement. However, the advanced architecture of Liquidsoap allows it to handle both audio and video. The main focus of this book will be audio streams, but this chapter is dedicated to handling video.

Video data. The first thing to remark is that if processing and transmitting audio requires handling large amounts of data, video requires processing *huge* amounts of data. A video in a decent resolution has 25 images per second at a resolution of 720p, which means 1280×720 pixels, each pixel consisting of three channels (generally, red, green and blue, or RGB for short) each of which is usually coded on one byte. This means that one second of uncompressed video data weights 65 MB, the equivalent of more than 6 minutes of uncompressed audio in CD quality! And these are only the minimal requirements for a video to be called HD (*High Definition*), which is the kind of video which is being watched everyday on the internet: in practice, even low-end devices can produce much higher resolutions than this.

This volume of data means that manipulation of video, such as combining videos or applying effects, should be coded very efficiently (by which we mean down to fine-tuning the assembly code for some parts), otherwise the stream generator will not be able to apply them in realtime on a standard recent computer. It also means that even copying of data should be avoided, the speed of memory accesses is also a problem at such rates.

A usual video actually consists of two streams: one for the video and one for the audio. We want to be able to handle them separately, so that we can apply all the operations specific to audio described in previous sections to videos, but the video and audio stream should be kept in perfect sync (even a very small delay between audio is video can be noticed).

File formats. We have seen that there is quite a few compressed formats available for audio and the situation is the same for video, but the video codecs generally involve many configuration options exploiting specificities of video, such as the fact two consecutive images in a video are usually quite similar. Fortunately, most of the common formats are handled by high-level libraries such as *FFmpeg*. This solves the problem for decoding, but for encoding we are still left with many parameters to specify, which can have a large impact

on the quality of the encoded video and on the speed of the compression (finding the good balance is somewhat of an art).

Video effects. As for audio, many manipulations of video files are expected to be present in a typical workflow.

- *Fading*: as for audio tracks, we should be able to fade between successive videos, this can be a smooth fade, or one video slides on top of the other, and so on.
- *Visual identity*: we should be able to add the logo of our channel, add a sliding text at the bottom displaying the news or listing the shows to come.
- *Color grading*: as for audio tracks, we should be able to give a particular ambiance by having uniform colors and intensities between tracks.

3 Installation

In order to install Liquidsoap you should either download compiled binaries for your environment, or compile it by yourself. The latest is slightly more involved, although it is a mostly automated process, but it allows to easily obtain a cutting-edge version and take part of the development process. These instructions are for the latest released version at the time of the writing, you are encouraged to consult the online documentation.

3.1 Automated building using opam

The recommended method to install Liquidsoap is by using the package manager opam. This program, which is available on all major distributions and architectures, makes it easy to build programs written in OCaml by installing the required dependencies (the libraries the program needs to be compiled) and managing consistency between various versions (in particular, it takes care of recompiling all the affected programs when a library is installed or updated). Any user can install packages with opam, no need to be root: the files it installs are stored in a subdirectory of the home directory, named `.opam`. The opam packages for Liquidsoap and associated libraries are actively maintained.

Installing opam. The easiest way to install opam on any achitecture is by running the command

```
sh <(curl -sL https://git.io/fjMth)
```

or by installing the `opam` package with the package manager of your distribution, e.g., for Ubuntu,

```
sudo apt install opam
```

or by downloading the binaries from the opam website. In any case, you should ensure that you have at least the version 2.0.0 of opam: the version number can be checked by running `opam --version`.

If you are installing opam for the first time, you should initialize the list of opam packages with

```
opam init
```

You can answer yes to all the questions it asks (if it complains about the absence of `bwrap`, either install it or add the flag `--disable-sandboxing` to the above command line). Next thing, you should install a recent version of the OCaml compiler by running

```
opam switch create 4.13.0
```

It does take a few minutes, because it compiles OCaml, so get prepared to have a coffee.

Installing Liquidsoap. Once this is done, a typical installation of Liquidsoap with support for MP3 encoding/decoding and Icecast is done by executing:

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

The first line (`opam depext ...`) takes care of installing the required external dependencies, i.e., the libraries we are relying on, but did not develop by ourselves. Here, we want to install the dependencies required by `taglib` (the library to read tags in audio files), `mad` (to decode MP3), `lame` (to encode MP3), `cry` (to stream to Icecast), `samplerate` (to resample audio) and finally `liquidsoap`. The second line (`opam install ...`) actually install the libraries and programs. Here also, the compilation takes some time (around a minute on a recent computer).

Most of Liquidsoap's dependencies are only optionally installed by opam. For instance, if you want to enable ogg/vorbis encoding and decoding after you've already installed Liquidsoap, you should install the `vorbis` library by executing:

```
opam depext  vorbis
opam install vorbis
```

Opam will automatically detect that this library can be used by Liquidsoap and will recompile it which will result in adding support for this format in Liquidsoap. The list of all optional dependencies that you may enable in Liquidsoap can be obtained by typing

```
opam info liquidsoap
```

and is detailed below.

Installing the cutting-edge version. The version of Liquidsoap which is packaged in opam is the latest release of the software. However, you can also install the cutting-edge version of Liquidsoap, for instance to test upcoming features. Beware that it might not be as stable as a release, although this is generally the case: our policy enforces that the developments in progress are performed apart, and integrated into the main branch only once they have been tested and reviewed.

In order to install this version, you should first download the repository containing all the code, which is managed using the git version control system:

```
git clone https://github.com/savonet/liquidsoap.git
```

This will create a `liquidsoap` directory with the sources, and you can then instruct opam to install Liquidsoap from this directory with the following commands:

```
opam pin add liquidsoap .
```

From time to time you can update your version by downloading the latest code and then asking opam to rebuild Liquidsoap:

```
git pull
opam upgrade liquidsoap
```

Updating libraries. If you also need a recent version of the libraries in the Liquidsoap ecosystem, you can download all the libraries at once by typing

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

You can then update a given library (say, `ocaml-ffmpeg`) by going in its directory and pinning it with opam, e.g.

```
cd ocaml-ffmpeg
opam pin add .
```

(and answer yes if you are asked questions).

3.2 Using binaries

If you want to avoid compiling Liquidsoap, or if opam is not working on your platform, the easiest way is to use precompiled binaries of Liquidsoap, if available.

Linux. There are packages for Liquidsoap in most Linux distributions. For instance, in Ubuntu or Debian, the installation can be performed by running

```
sudo apt install liquidsoap
```

which will install the `liquidsoap` package, containing the main binaries. It comes equipped with most essential features, but you can install plugins in the packages `liquidsoap-plugin-...` to have access to more libraries; for instance, installing `liquidsoap-plugin-flac` will add support for the FLAC lossless audio format or `liquidsoap-plugin-all` will install all available plugins (which might be a good idea if you are not sure about which you are going to need).

macOS. No binaries are provided for macOS, the preferred method is opam, see above.

Windows. Pre-built binaries are provided on the releases pages in a file with a name of the form `liquidsoap-vN.N.N-win64.zip`. It contains directly the program, no installer is provided at the moment.

3.3 Building from source

In some cases, it is necessary to build directly from source (e.g., if opam is not supported on your exotic architecture or if you want to modify the source code of Liquidsoap). This can be a difficult task, because Liquidsoap relies on an up-to-date version of the OCaml compiler, as well as a bunch of OCaml libraries and, for most of them, corresponding C library dependencies.

Installing external dependencies. In order to build Liquidsoap, you first need to install the following OCaml libraries: `ocamlfind`, `sedlex`, `menhir`, `pcre` and `camomile`. You can install those using your package manager

```
sudo apt install ocaml-findlib libsedlex-ocaml-dev menhir libpcre-ocaml-dev
↪  libcamomile-ocaml-dev
```

(as you can remark, OCaml packages for Debian or Ubuntu often bear names of the form `libxxx-ocaml-dev`), or using opam

```
opam install ocamlfind sedlex menhir pcre camomile
```

or from source.

Getting the sources of Liquidsoap. The sources of Liquidsoap, along with the required additional OCaml libraries we maintain can be obtained by downloading the main git repository, and then run scripts which will download the submodules corresponding to the various libraries:

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

Installing. Next, you should copy the file `PACKAGES.default` to `PACKAGES` and possibly edit it: this file specifies which features and libraries are going to be compiled, you can add/remove those by uncommenting/commenting the corresponding lines. Then, you can generate the `configure` scripts:

```
./bootstrap
```

and then run them:

```
./configure
```

This script will check that whether the required external libraries are available, and detect the associated parameters. It optionally takes parameters such as `--prefix` which can be used to specify in which directory the installation should be performed. You can now build everything

```
make
```

and then proceed to the installation

```
make install
```

You may need to be root to run the above command in order to have the right to install in the usual directories for libraries and binaries.

3.4 Docker image

Docker images are provided as `savonet/liquidsoap`: these are Debian-based images with Liquidsoap pre-installed (and not much more in order to have a file as small as possible), which you can use to easily and securely deploy scripts using it. The tag `main` always contains the latest version, and is automatically generated after each modification.

We refer the reader to the Docker documentation for the way such images can be used. For instance, you can have a shell on such an image with

```
docker run -it --entrypoint /bin/bash savonet/liquidsoap:main
```

By default, the docker image does not have access to the soundcard of the local computer (but it can still be useful to stream over the internet for instance). It is however possible to bind the ALSA soundcard of the host computer inside the image. For instance, you can play a sine (see there) by running:

```
docker run -it -v /dev/snd:/dev/snd --privileged savonet/liquidsoap:main
↪  liquidsoap 'output.alsa(sine())'
```

This single line should work on any computer on which Docker is installed: no need to install opam, various libraries, or Liquidsoap, it will automatically download for you an image where all this is pre-installed.

3.5 Libraries used by Liquidsoap

We list below some of the libraries which can be used by Liquidsoap. They are detected during the compilation of Liquidsoap and, in this case, support for the libraries is added. We recall that a library `ocaml-something` can be installed via opam with

```
sudo opam depext  something
sudo opam install something
```

which will automatically trigger a rebuild of Liquidsoap, as explained in the above section.

General. Those libraries add support for various things:

- `camomile`: charset recoding in metadata (those are generally encoded in UTF-8 which can represent all characters, but older files used various encodings for characters which can be converted),
- `ocaml-inotify`: getting notified when a file changes (e.g. for reloading a playlist when it has been updated),
- `ocaml-magic`: file type detection (e.g. this is useful for detecting that a file is an MP3 even if it does not have the `.mp3` extension),
- `ocaml-lo`: OSC (Open Sound Control) support for controlling the radio (changing the volume, switching between sources) via external interfaces (e.g. an application on your phone),
- `ocaml-ssl`: SSL support for connecting to secured websites (using the HTTPS protocol),
- `ocurl`: downloading files over HTTP,
- `osx-secure-transport`: SSL support via OSX's SecureTransport,
- `yojson`: parsing JSON data (useful to exchange data with other applications).

Input / output. Those libraries add support for using soundcards for playing and recording sound:

- `ocaml-alsa`: soundcard input and output with ALSA,
- `ocaml-ao`: soundcard output using AO,
- `ocaml-ffmpeg`: input and output over various devices,
- `ocaml-gstreamer`: input and output over various devices,
- `ocaml-portaudio`: soundcard input and output,
- `ocaml-pulseaudio`: soundcard input and output.

Among those, ALSA is very low level and is probably the one you want to use in order to minimize latencies. Other support a wider variety of soundcards and usages.

Other outputs:

- `ocaml-cry`: output to Icecast servers,
- `ocaml-bjack`: JACK support for virtually connecting audio programs,
- `ocaml-lastfm`: Last.fm scrobbling (this website basically records the songs you have listened),
- `ocaml-srt`: transport over network using SRT protocol.

Sound processing. Those add support for sound manipulation:

- `ocaml-dssi`: sound synthesis plugins,
- `ocaml-ladspa`: sound effect plugins,
- `ocaml-lilv`: sound effect plugins,
- `ocaml-samplerate`: samplerate conversion in audio files,
- `ocaml-soundtouch`: pitch shifting and time stretching.

Audio file formats.

- `ocaml-faad`: AAC decoding,
- `ocaml-fdkaac`: AAC+ encoding,
- `ocaml-ffmepg`: encoding and decoding of various formats,
- `ocaml-flac`: Flac encoding and decoding,
- `ocaml-gstreamer`: encoding and decoding of various formats,
- `ocaml-lame`: MP3 encoding,
- `ocaml-mad`: MP3 decoding,
- `ocaml-ogg`: Ogg containers,
- `ocaml-opus`: Ogg/Opus encoding and decoding,
- `ocaml-shine`: fixed-point MP3 encoding,
- `ocaml-speex`: Ogg/Speex encoding and decoding,
- `ocaml-taglib`: MP3 metadata decoding,
- `ocaml-vorbis`: Ogg/Vorbis encoding and decoding.

Playlists.

- `ocaml-xmlplaylist`: support for playlist formats based on XML.

Video. Video conversion:

- `ocaml-ffmpeg`: video conversion,
- `ocaml-gavl`: video conversion,
- `ocaml-theora`: Ogg/Theora encoding and decoding.

Other video-related libraries:

- `camlimages`: decoding of various image formats,
- `gd4o`: rendering of text,
- `ocaml-frei0r`: video effects,
- `ocaml-sdl`: display, text rendering and image formats.

Runtime dependencies. Those optional dependencies can be used by Liquidsoap if installed, they are detected at runtime and do not require any particular support during compilation:

- `awscli`: `s3://` and `polly://` protocol support for Amazon web servers,
- `curl`: downloading files with `http`, `https` and `ftp` protocols,
- `ffmpeg`: external input and output, `replay_gain`, level computation, and more,
- `youtube-dl`: YouTube video and playlist downloading support.

4 Setting up a simple radio station

4.1 The sound of a sine wave

Our first sound. In order to test your installation, you can try the following in a console:

```
liquidsoap 'output(sine())'
```

This instructs Liquidsoap to run the program

```
output(sine())
```

which plays a sine wave at 440 Hertz. The operator `sine` is called a *source*: it generates audio (here, a sine wave) and `output` is an operator which takes a source as parameter and plays it on the soundcard. When running this program, you should hear the expected well-known sound and see lots of lines looking like this:

```
2021/02/18 15:20:44 >>> LOG START
2021/02/18 15:20:43 [main:3] Liquidsoap 2.0.0
...
```

These are the *logs* for Liquidsoap, which are messages describing what each operator is doing. These are often useful to follow what the script is doing, and contain important information in order to understand what is going wrong if it is the case. Each of these lines begins with the date and the hour the message was issued, followed by who emitted the message (i.e. which operator), its importance, and the actual message. For instance, `[main:3]` means that the main process of Liquidsoap emitted the message and that its importance is `3`. The lower the number is, the more important the message is: `1` is a critical message (the program might crash after that), `2` a severe message (something that might affect the program in a deep way), `3` an important message, `4` an information and `5` a debug message (which can generally be ignored). By default, only messages with importance up to `3` are displayed.

Scripts. You will soon find out that a typical radio takes more than one line of code, and it is not practical to write everything on the command line. For this reason, the code for describing your webradio can also be put in a *script*, which is a file containing all the code for your radio. For instance, for our sine example, we can put the following code in a file `radio.liq`:

```
#!/usr/bin/env liquidsoap
# Let's play a sine wave
output(sine())
```

The first line says that the script should be executed by Liquidsoap. It begins by `#!` (sometimes called a *shebang*) and then says that `/usr/bin/env` should be used in order to find the path for the `liquidsoap` executable. If you know its complete path (e.g. `/usr/bin/liquidsoap`) you could also directly put it:

```
#!/usr/bin/liquidsoap
```

In the rest of the book, we will generally omit this first line, since it is always the same. The second line of `radio.liq`, is a comment. You can put whatever you want here: as long as the line begins with `#`, it will not be taken in account. The last line is the actual program we already saw above.

In order to execute the script, you should ensure that the program is executable with the command

```
chmod +x radio.liq
```

and you can then run it with

```
./radio.liq
```

which should have the same effect as before. Alternatively, the script can also be run by passing it as an argument to Liquidsoap

```
liquidsoap radio.liq
```

in which case the first line (starting with `#!`) is not required.

Variables. In order to have more readable code, one can use variables which allow giving names to sources. For instance, we can give the name `s` to our sine source and then play it. The above code is thus equivalent to

```
s = sine()
output(s)
```

Parameters. In order to investigate further the possible variations on our example, let us explore the parameters of the `sine` operator. In order to obtain detailed help about this operator, we can type, in a console,

```
liquidsoap -h sine
```

which will output

```
Generate a sine wave.

Type: (?id : string, ?amplitude : float, ?float) -> source(audio=internal('a),
↪  video=internal('b), midi=internal('c))

Category: Source / Input

Parameters:

 * id : string (default: "")
     Force the value of the source ID.
```

```
 * amplitude : float (default: 1.0)
     Maximal value of the waveform.

 * (unlabeled) : float (default: 440.0)
     Frequency of the sine.
```

(this information is also present in the online documentation).

It begins with a description of the operator, followed by its type, category and arguments (or parameters). There is also a section for methods, which is not shown above, but we simply ignore it for now, it will be detailed in a subsequent section. Here, we see in the type that it is a function, because of the presence of the arrow "->": the type of the arguments is indicated on the left of the arrow and the type of the output is indicated on the right. More precisely, we see that it takes three arguments and returns a source with any number of audio, video and MIDI channels (the precise meaning of `source` is detailed in this section). The three arguments are indicated in the type and detailed in the following `Parameters` section:

- the first argument is a string labeled `id`: this is the name which will be displayed in the logs,
- the second is a float labeled `amplitude`: this controls how loud the generated sine wave will be,
- the third is a float with no label: the frequency of the sine wave.

All three arguments are optional, which means that a default value is provided and will be used if it is not specified. This is indicated in the type by the question mark "?" before each argument, and the default value is indicated in `Parameters` (e.g. the default amplitude is `1.0` and the default frequency is `440.` Hz).

If we want to generate a sine wave of 2600 Hz with an amplitude of 0.8, we can thus write

```
s = sine(id="my_sine", amplitude=0.8, 2600.)
output(s)
```

Note that the parameter corresponding to id has a label `id`, which we have to specify in order to pass the corresponding argument, and similarly for amplitude, whereas there is no label for the frequency.

Finally, just for fun, we can hear an A minor chord by adding three sines:

```
s1 = sine()
s2 = sine(440. * pow(2., 3. / 12.))
s3 = sine(440. * pow(2., 7. / 12.))
s  = add([s1, s2, s3])
output(s)
```

We generate three sines at frequencies 440 Hz, $440 \times 2^{3/12}$ Hz and $440 \times 2^{7/12}$ Hz, adds them, and plays the result. The operator `add` is taking as argument a *list* of sources, delimited by square brackets, which could contain any number of elements.

4.2 A radio

Playlists and more. Since we are likely to be here not to make synthesizers but rather radios, we should start playing actual music instead of sines. In order to do so, we have the `playlist` operator which takes as argument a *playlist*: it can be a file containing paths to

audio files (WAV, MP3, etc.), one per line, or a playlist in a standard format (pls, m3u, xspf, etc.), or a directory (in which case the playlist consists of all the files in the directory). For instance, if our music is stored in the `~/Music` directory, we can play it with

```
s = playlist("~/Music")
output(s)
```

As usual, the operator `playlist` has a number of interesting optional parameters which can be discovered with `liquidsoap -h playlist`. For instance, by default, the files are played in a random order, but if we want to play them as indicated in the list we should pass the argument `mode="normal"` to `playlist`. Similarly, if we want to reload the playlist whenever it is changed, the argument `reload_mode="watch"` should be passed.

A playlist can refer to distant files (e.g. urls of the form `http://path/to/file.mp3`) in which case they are going to be downloaded beforehand. If you want to use a live stream, which can be very long or even infinite, the operator `input.http` should be used instead:

```
s = input.http("https://icecast.radiofrance.fr/fip-hifi.aac")
output(s)
```

The playlist can also mention special sort of files, using particular *protocols* which are proper to Liquidsoap: those do not refer to actual files, but rather describe how to produce files. For instance, a line of the form

```
say:Hello everybody!
```

in a playlist will instruct Liquidsoap to use a text-to-speech program in order to generate a file in which "Hello everybody!" is pronounced.

Finally, there are other types of inputs. For instance, the operator `input.alsa` can be used to capture the sound of a microphone on a soundcard, with the ALSA library. This means that you should be able to hear your voice with

```
s = buffer(input.alsa())
output(s)
```

The ALSA input and the output each have their own way of synchronizing with time: in our terminology, we say that they have different *clocks*, see a later section. This will be detected by Liquidsoap an a script such as

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

will be rejected. This is the reason why we need to use the `buffer` operator here which will compute part of the input stream in advance (1 second by default) and will therefore be able to cope with small discrepancies in the way the operators synchronize. If you try the above example, you can hear that there is a slight delay between your voice and the output due to the buffering.

Fallible sources and fallbacks. Some sources are not always available, and we say that such a source is *fallible*. A typical example is a source obtained by `input.http`: at some point the stream might stop (e.g. if it is only available during daytime), or be subject to technical difficulties (e.g. it gets disconnected from the internet for a short period of time). In this case, we generally want to fall back to another source, typically an emergency playlist consisting of local files which we are sure are going to be available. This can be achieved by using the `fallback` operator which plays the first available source in a list of sources:

```
stream    = input.http("http://...")
emergency = playlist("~/Music")
```

```
s         = fallback([stream, emergency])
output(s)
```

This means that `s` will have the same contents as `stream` if it is available, and as `emergency` otherwise.

Fallibility detection. Liquidsoap automatically detects that a source is fallible and issues an error if this is not handled, by a `fallback` for instance, in order to make sure that we will not unexpectedly have nothing to stream at some point. We did not see this up to now because `output` is an advanced operator which automatically uses silence as fallback, because it is primarily intended for quick and dirty checking of the stream. However, if we use the primitive functions for outputting audio, we will be able to observe this behavior. For instance, if we try to use the operator `output.pulseaudio`, which plays a source on a soundcard using the PulseAudio library,

```
s = input.http("http://...")
output.pulseaudio(s)
```

we obtain the following error:

```
At line 1, char 4-27:
Error 7: Invalid value: That source is fallible
```

This means that Liquidsoap has detected that the source declared at line 1 from character 4 to character 27, i.e. the `input.http`, is fallible. We could simply ignore this warning, by passing the parameter `fallible=true` to the `output.pulseaudio` operator, but the proper way to fix this consists in having a fallback to a local file:

```
s         = input.http("http://...")
emergency = single("/radio/emergency.mp3")
s         = fallback(track_sensitive=false, [s, emergency])
output.pulseaudio(s)
```

Note that we are using `single` here instead of `playlist`: this operator plays a single file and ensures that the file is available before running the script so that we know it will not fail. The argument `track_sensitive=false` means that we want to get back to the live stream as soon as it is available again, otherwise it would wait the end of the track for switching back from emergency playlist to the main radio. Also remark that we are defining `s` twice: this is not a problem at all, whenever we reference `s`, the last definition is used, otherwise said the second definition replaces the first.

Falling back to blank. Another option to make the stream infallible would be to fall back to silence, which in Liquidsoap can be generated with the operator `blank`:

```
s = input.http("http://...")
s = fallback(track_sensitive=false, [s, blank()])
output.pulseaudio(s)
```

This behavior is so common that Liquidsoap provides the `mksafe` function which does exactly that:

```
s = buffer(input.http("http://..."))
s = mksafe(s)
output.pulseaudio(s)
```

Streams depending on the hour. A typical radio will do some scheduling, typically by having different playlists at different times of the day. In Liquidsoap, this is achieved by using the `switch` operator: this operator takes a list of pairs consisting of a predicate (a function returning a boolean `true` or `false`) and a source, and plays the first source for which the predicate is true. For time, there is a special syntax:

```
{ 8h-20h }
```

is a predicate which is true when the current time is between 8h and 20h (or 8 am and 8 pm if you like this better). This means that if we have two playlists, one for the day and one for the night, and want a live show between 19h and 20h, we can set this up as follows:

```
day   = playlist("/radio/day.pls")   # Day music
night = playlist("/radio/night.pls") # Night music
mic   = buffer(input.alsa())         # Microphone
radio = switch([({8h-19h}, day), ({19h-20h}, mic), ({20h-8h}, night)])
```

By default, the `switch` operator will wait for the end of the track of a source before switching to the next one, but immediate switching can be achieved by adding the argument `track_sensitive=false`, as for the `fallback` operator.

Jingles. The next thing we want to be able to do is to insert jingles. We suppose that we have a playlist consisting of all the jingles of our radio and we want to play roughly one jingle every 5 songs. This can be achieved by using the `random` operator:

```
jingles = playlist("/radio/jingles.pls")
radio   = random(weights=[1, 4], [jingles, radio])
```

This operator randomly selects a track in a list of sources each time a new track has to be played (here this list contains the jingles playlist and the radio defined above). The `weight` argument says how many tracks of each source should be taken in average: here we want to take 1 jingle for 4 radio tracks. The selection is randomized however and it might happen that two jingles are played one after the other, although this should be rare. If we want to make sure that we play 1 jingle and then exactly 4 radio songs, we should use the `rotate` operator instead:

```
radio = rotate(weights=[1, 4], [jingles, radio])
```

Crossfading. Now that we have our basic sound production setup, we should try to make things sound nicer. A first thing we notice is that the transition between songs is quite abrupt whereas we would rather have a smooth chaining between two consecutive tracks. This can be addressed using the `crossfade` operator which will take care of this for us. If we insert the following line

```
radio = crossfade(fade_out=3., fade_in=3., duration=5., radio)
```

at each end of track the song will fade out during 3 seconds, the next track will fade in for 3 seconds and the two will overlap during 5 seconds, ensuring a pleasant transition.

Audio effects. In order to make the sound more uniform, we can use plugins. For instance, the `normalize` operator helps you to have a uniform volume by dynamically changing it, so that volume difference between songs is barely heard:

```
radio = normalize(radio)
```

In practice, it is better to precompute the gain of each audio track in advance and change the volume according to this information, often called *ReplayGain*, see there. There are also various traditional sound effects that can be used in order to improve the overall color and personality of the sound. A somewhat reasonable starting point is provided by the `nrj` operator:

```
radio = nrj(radio)
```

Many more details about sound processing are given in there.

Icecast output. Now that we have set up our radio, we could play it locally by adding

```
output(radio)
```

at the end of the script, but we would rather stream it to the world instead of having it only on our speakers.

Installing Icecast. In order to do so, we first need to set up an Icecast server which will relay the stream to users connecting to it. The way you should proceed with its installation depends on your distribution, for instance on Ubuntu you can type

```
sudo apt install icecast2
```

The next thing we should do is to modify the configuration which is generally located in the file `/etc/icecast2/icecast.xml`. In particular, we should modify the lines

```
<source-password>hackme</source-password>
 <relay-password>hackme</relay-password>
 <admin-password>hackme</admin-password>
```

which are the passwords for sources (e.g. the one Liquidsoap is going to use in order to send its stream to Icecast), for relays (used when relaying a stream, you are not going to use this one now but still want to change the password) and for the administrative interface. By default, all three are `hackme`, and we will use that in our examples, but, again, you should change them in order not to be hacked. Have a look at other parameters though, they are interesting too! Once the configuration modified, you should the restart Icecast with the command

```
sudo /etc/init.d/icecast2 restart
```

If you are on a system such as Ubuntu, the default configuration prevents Icecast from running, because they want to ensure that you have properly configured it. In order to enable it, before restarting, you should set

```
ENABLE=true
```

at the end of the file `/etc/default/icecast2`. More information about setting up Icecast can be found on its website.

Icecast output. Once this is set up, you should add the following line to your script in order to instruct Liquidsoap to send the stream to Icecast:

```
output.icecast(%mp3, host="localhost", port=8000,
               password="hackme", mount="my-radio.mp3", radio)
```

The parameters of the operator `output.icecast` we used here are

- the format of the stream: here we encode as MP3,

- the parameters of your Icecast server: hostname, port (8000 is the default port) and password for sources,
- the mount point: this will determine the URL of your stream,
- and finally, the source we want to send to Icecast, `radio` in our case.

If everything goes on well, you should be able to listen to your radio by going to the URL

```
http://localhost:8000/my-radio.mp3
```

with any modern player or browser. If you want to see the number of listeners of your stream and other useful information, you should have a look at the stats of Icecast, which are available at

```
http://localhost:8000/admin/stats.xsl
```

with the login for administrators (`admin` / `hackme` by default).

The encoder. The first argument `%mp3`, which controls the format, is called an *encoder* and can itself be passed some arguments in order to fine tune the encoding. For instance, if we want our MP3 to have a 256k bitrate, we should pass `%mp3(bitrate=256)`. It is perfectly possible to have multiple streams with different formats for a single radio: if we want to also have an AAC stream we can add the line

```
output.icecast(%fdkaac, host="localhost", port=8000,
               password="hackme", mount="my-radio.aac", radio)
```

By the way, support for AAC is not built in in the default installation. If you get the message

```
Error 12: Unsupported format!
You must be missing an optional dependency.
```

this means that you did not enable it. In order to do so in an opam installation, you should type

```
opam depext  fdkaac
opam install fdkaac
```

Summing up. The typical radio script we arrived at is the following one:

```
#!/bin/env liquidsoap
# Set up the playlists
day   = playlist("/radio/day.pls")   # Day music
night = playlist("/radio/night.pls") # Night music
mic   = buffer(input.alsa())         # Microphone
radio = switch([({8h-19h}, day), ({19h-20h}, mic), ({20h-8h}, night)])
# Add crossfading
radio = crossfade(fade_out=3., fade_in=3., duration=5., radio)
# Add jingles
jingles = playlist("/radio/jingls.pls")
radio = random(weights=[1, 4], [jingles, radio])
# Add some audio effects
radio = nrj(normalize(radio))
# Just in case, a fallback
radio = fallback([radio, single("fallback.mp3")])
# Output to icecast both in mp3 and aac
output.icecast(%mp3,    host="localhost", port=8000, password="hackme",
```

```
                mount="my-radio.mp3", radio)
output.icecast(%fdkaac, host="localhost", port=8000, password="hackme",
                mount="my-radio.aac", radio)
```

That's it for now, we will provide many more details in this chapter.

5 A programming language

Before getting into the more advanced radio setups which can be achieved with Liquidsoap, we need to understand the language and the general concepts behind it. If you are eager to start your radio, it might be a good idea to at least skim though this chapter quickly at a first reading, and come back later to it when a deeper knowledge about a specific point is required.

5.1 General features

Liquidsoap is a novel language which was designed from scratch. We present the generic constructions, feature specifically related to streaming are illustrated in next chapter and further detailed in this chapter.

Typing. One of the main features of the language is that it is *typed*. This means that every expression belongs to some type which indicates what it is. For instance, `"hello"` is a *string* whereas `23` is an *integer*, and, when presenting a construction of the language, we will always indicate the associated type. Liquidsoap implements a *typechecking* algorithm which ensures that whenever a string is expected a string will actually be given, and similarly for other types. This is done without running the program, so that it does not depend on some dynamic tests, but is rather enforced by theoretical considerations. Another distinguishing feature of this algorithm is that it also performs *type inference*: you never actually have to write a type, those are guessed automatically by Liquidsoap. This makes the language very safe, while remaining very easy to use. For curious people reading French, the algorithm and the associated theory are described in a publication (Baelde and Mimram 2008).

Incidentally, apart from the usual type information which can be found in many languages, Liquidsoap also uses typing to check the coherence of parameters which are specific to streaming. For instance, the number of audio channels of streams is also present in their type, and it ensures that operators always get the right number of channels.

Functional programming. The language is *functional*, which means that you can very easily define functions, and that functions can be passed as arguments of other functions. This might look like a crazy thing at first, but it is actually quite common in some language communities (such as OCaml). It also might look quite useless: why should we need such

functions when describing webradios? You will soon discover that it happens to be quite convenient in many places: for handlers (we can specify the function which describes what to do when some event occurs such as when a DJ connects to the radio), for transitions (we pass a function which describes the shape we want for the transition) and so on.

Streams. The unique feature of Liquidsoap is that it allows the manipulation of *sources* which are functions which will generate streams. These streams typically consist of stereo audio data, but we do restrict to this: they can contain audio with arbitrary number of channels, they can also contain an arbitrary number of video channels, and also MIDI channels (there is limited support for sound synthesis).

Execution model. When running a Liquidsoap program, the compiler goes through these four phases:

1. *lexical analysis* and *parsing*: Liquidsoap ingests your program and ensures that its syntax follows the rules,
2. *type inference* and *type checking*: Liquidsoap checks that your program does not contain basic errors and that types are correct,
3. *compilation* of the program: this produces a new program which will generate the stream (a *stream generator*),
4. *instantiation*: the sources are created and checked to be infallible where required,
5. *execution*: we run the stream generator to actually produce audio.

The two last phases can be resumed by the following fact: Liquidsoap is a *stream generator generator*, it generates stream generators (sic).

In order to illustrate this fact, consider the following script (don't worry if you don't understand all the details for now, it uses concepts which will be detailed below):

```
def note(n) = sine(440. * pow(2., n / 12.)) end
s = add(list.map(note, [0., 3., 7.]))
output(s)
```

Let us explain how this script should be thought of as a way of describing how to generate a stream generator. In order to construct the stream generator, Liquidsoap will execute the function `list.map` which will produce the list obtained by applying the function `note` on each element of the list and, in turn, this function will be replaced by its definition, which consists of a `sine` generator. The execution of the script will act as if Liquidsoap successively replaced the second line by

```
s = add([note(0.), note(3.), note(7.)])
```

and then by

```
s = add([sine(440. * pow(2., 0. / 12.)),
         sine(440. * pow(2., 3. / 12.)),
         sine(440. * pow(2., 7. / 12.))])
```

and finally by

```
s = add([sine(440.), sine(523.25), sine(659.26)])
```

which is the actual stream generator. We see that running the script has generated the three `sine` stream generators!

Standard library. Although the core of Liquidsoap is written in OCaml, many of the functions of Liquidsoap are written in the Liquidsoap language itself. Those are defined in the `stdlib.liq` script, which is loaded by default and includes all the libraries. You should not be frightened to have a look at the standard library, it is often useful to better grasp the language, learn design patterns and tricks, and add functionalities. Its location on your system is indicated in the variable `configure.libdir` and can be obtained by typing

```
liquidsoap --check "print(configure.libdir)"
```

5.2 Writing scripts

Choosing an editor. Scripts in Liquidsoap can be written in any text editor, but things are more convenient if there is some specific support. We have developed a mode for the Emacs editor which adds syntax coloration and indentation when editing Liquidsoap files. User-contributed support for Liquidsoap is also available for popular editors such as Visual Studio Code or vim.

Documentation of operators. When writing scripts you will often need details about a particular operator and its arguments. We recall from earlier that the documentation of an operator `operator`, including its type and a description of its arguments, can be obtained by typing

```
liquidsoap -h operator
```

This documentation is also available on the website.

Interactive mode. In order to test the functions that will be introduced in this section, it can be convenient to use the *interactive mode* of Liquidsoap which can be used to type small expressions and immediately see their result. This interactive mode is rarely used in practice, but is useful to learn the language and do small experiments. It can be started with

```
liquidsoap --interactive
```

It will display a "#", meaning it is waiting for expressions, which are programs in the language. They have to be ended by ";;" in order to indicate that Liquidsoap should evaluate them. For instance, if we type

```
name = "Sam";;
```

it answers

```
name : string = "Sam"
```

which means that we have defined a variable `name` whose type is `string` and whose value is `"Sam"`. It can be handy as a calculator:

```
2*3;;
```

results in

```
- : int = 6
```

("-" means that we did not define a variable, that the type of the expression is `int` and that it evaluates to 6). Also, variables can be reused: if we type

```
print("Hello #{name} and welcome!");;
```

it will answer

```
Hello Sam and welcome!
- : unit = ()
```

The command `print` was evaluated and displays its argument and then the result is shown, in the same format as above: `-` means that we did not define a variable, the type of the result is `unit` and its value is `()`. The meaning of these is detailed below. In the following, all examples starting by `#` indicate that they are being entered in the interactive mode.

Inferred types. Another useful feature is the `-i` option of Liquidsoap which displays the types of variables in a file. For instance, if we have a file `test.liq` containing

```
x = 3.2
def f (x) = x + 1 end
```

and we run

```
liquidsoap -i test.liq
```

it will display the types for `x` and `f`:

```
x : float
f : (int) -> int
```

meaning that `x` is a floating point number and `f` is a function taking an integer as argument and returning an integer.

5.3 Basic values

We begin by describing the values one usually manipulates in Liquidsoap.

Integers and floats. The *integers*, such as `3` or `42`, are of type `int`. Depending on the current architecture of the computer on which we are executing the script (32 or 64 bits, the latter being the most common nowadays) they are stored on 31 or 63 bits. The minimal (resp. maximal) representable integer can be obtained as the constant `min_int` (resp. `max_int`); typically, on a 64 bits architecture, they range from -4611686018427387904 to 4611686018427387903.

The *floating point numbers*, such as `2.45`, are of type `float`, and are in double precision, meaning that they are always stored on 64 bits. We always write a decimal point in them, so that `3` and `3.` are not the same thing: the former is an integer and the latter is a float. This is a source of errors for beginners, but is necessary for typing to work well. For instance, if we try to execute a program containing the instruction

```
s = sine(500)
```

it will raise the error

```
At line 1, char 9:
Error 5: this value has type int but it should be a subtype of float
```

which means that the sine function expects a float as argument, but an integer is provided. The fix here obviously consists in replacing "`500`" by "`500.`" (beware of the dot).

The usual arithmetic operations are available (`+`, `-`, `*`, `/`), and work for both integers and floats. For floats, traditional arithmetic functions are available such as `sqrt` (square root), `exp` (exponential), `sin` (sine), `cos` (cosine) and so on. Random integers and floats can be generated with the `random.int` and `random.float` functions.

Strings. Strings are written between double or single quotes, e.g. `"hello!"` or `'hello!'`, and are of type `string`.

The function to output strings on the standard output is `print`, as in

```
print("Hello, world!")
```

Incidentally, this function can also be used to display values of any type, so that

```
print(3+2)
```

will display `5`, as expected. In practice, one rarely does use this functions, which displays on the standard output, but rather the logging functions `log.critical`, `log.severe`, `log.important`, `log.info` and `log.debug` which write strings of various importance in the logs, so that it is easier to keep track of them: they are timestamped, they can easily be stored in files, etc.

In order to write the character "`"`" in a string, one cannot simply type "`"`" since this is already used to indicate the boundaries of a string: this character should be *escaped*, which means that the character "`\`" should be typed first so that

```
print("My name is \"Sam\"!")
```

will actually display "`My name is "Sam"!`". Other commonly used escaped characters are "`\\`" for backslash and "`\n`" for new line. Alternatively, one can use the single quote notation, so that previous example can also be written as

```
print('My name is "Sam"!')
```

This is most often used when testing JSON data which can contain many quotes or for command line arguments when calling external scripts. The character "`\`" can also be used at the end of the string to break long strings in scripts without actually inserting newlines in the strings. For instance, the script

```
print("His name is \
       Romain.")
```

will actually print

```
His name is Romain.
```

Note that there is no line change between "is" and "Romain", and the indentation before "Romain" is not shown either.

The concatenation of two strings is achieved by the infix operator "`^`", as in

```
user = "dj"
print("Current user is " ^ user)
```

Instead of using concatenation, it is often rather convenient to use *string interpolation*: in a string, `#{e}` is replaced by the string representation of the result of the evaluation of the expression `e`:

```
user = "admin"
print("The user #{user} has just logged.")
```

will print `The user admin has just logged.` or

```
print("The number #{random.float()} is random.")
```

will print `The number 0.663455738438 is random.` (at least it did last time I tried).

The string representation of any value in Liquidsoap can be obtained using the function `string_of`, e.g. `string_of(5)` is `"5"`. Some other useful string-related function are

- `string.length`: compute the length of a string

  ```
  # string.length("abc");;
  - : int = 3
  ```

- `string.sub`: extract a substring

  ```
  # string.sub("hello world!", start=6, length=5);;
  - : string = "world"
  ```

- `string.split`: split a string on a given character

  ```
  # string.split(separator=":", "a:42:hello");;
  - : [string] = ["a", "42", "hello"]
  ```

- `string.contains`: test whether a string contains (or begins or ends with) a particular substring,
- `string.quote`: escape shell special characters (you should always use this when passing strings to external programs).

Finally, some functions operate on *regular expressions*, which describe some shapes for strings:

- `string.match`: test whether a string matches a regular expression,
- `string.replace`: replace substrings matching a regular expression.

A regular expression `R` or `S` is itself a string where some characters have a particular meaning:

- `.` means "any character",
- `R*` means "any number of times something of the form `R`",
- `R|S` means "something of the form `R` or of the for `S`",

other characters represent themselves (and special characters such as `.`, `*` or `.` have to be escaped, which means that `\.` represents the character `.`). An example is worth a thousand words: we can test whether a string `fname` corresponds to the name of an image file with

```
string.match(pattern=".*\\.png|.*\\.jpg", fname)
```

Namely, this function will test if `fname` matches the regular expression `.*\.png|.*\.jpg` which means "any number of any character followed by `.png` or any number of any character followed by `.jpg`".

Booleans. The *booleans* are either `true` or `false` and are of type `bool`. They can be combined using the usual boolean operations

- `and`: conjunction,
- `or`: disjunction,
- `not`: negation.

Booleans typically originate from comparison operators, which take two values and return booleans:

- `==`: compares for equality,
- `!=`: compares for inequality,

- <=: compares for inequality,

and so on (<, >=, >). For instance, the following is a boolean expression:

```
(n < 3) and not (s == "hello")
```

The time predicates such as `10h-15h` are also booleans, which are true or false depending on the current time, see there.

Conditional branchings execute code depending on whether a condition is true or not. For instance, the code

```
if (1 <= x and x <= 12) or (not 10h-15h) then
  print("The condition is satisfied")
else
  print("The condition ain't satisified")
end
```

will print that the condition is satisfied when either `x` is between 1 and 12 or the current time is not between 10h and 15h. A conditional branching might return a value, which is the last computed value in the chosen branch. For instance,

```
y = if x < 3 then "A" else "B" end
```

will assign `"A"` or `"B"` to `y` depending on whether `x` is below 3 or not. The two branches of a conditional should always have the same return type:

```
x = if 1 == 2 then "A" else 5 end
```

will result in

```
At line 1, char 19-21:
Error 5: this value has type (...) -> string
but it should be a subtype of (...) -> int
```

meaning that `"A"` is a string but is expected to be an integer because the second branch returns an integer, and the two should be of same nature. The `else` branch is optional, in which case the `then` branch should be of type `unit`:

```
if x == "admin" then print("Welcome admin") end
```

In the case where you want to perform a conditional branching in the **`else`** branch, the **`elsif`** keyword should be used, as in the following example, which assigns 0, 1, 2 or 3 to `s` depending on whether `x` is `"a"`, `"b"`, `"c"` or something else:

```
s = if x == "a" then 0
    elsif x == "b" then 1
    elsif x == "c" then 2
    else 3 end
```

This is equivalent (but shorter to write) to the following sequence of imbricated conditional branchings:

```
s = if x == "a" then 0
    else
      if x == "b" then 1
      else
        if x == "c" then 2
        else 3 end
```

```
    end
  end
```

Finally, we should mention that the notation `c?a:b` is also available as a shorthand for `if c then a else b end`, so that the expression

```
y = if x < 3 then "A" else "B" end
```

can be shortened to

```
y = (x<3)?"A":"B"
```

(and people will think that you are a cool guy).

Unit. Some functions, such as `print`, do not return a meaningful value: we are interested in what they are doing (e.g. printing on the standard output) and not in their result. However, since typing requires that everything returns something of some type, there is a particular type for the return of such functions: `unit`. Just as there are only two values in the booleans (`true` and `false`), there is only one value in the unit type, which is written `()`. This value can be thought of as the result of the expression saying "I'm done".

In *sequences* of instructions, all the instructions but the last should be of type unit. For instance, the following function is fine:

```
def f()
  print("hello")
  5
end
```

This is a function printing "hello" and then returning 5, see below for details about functions. Sequences of instructions are delimited by newlines, but can also be separated by `;` in order to have them fit on one line, i.e., the above can equivalently be written

```
def f() = print("hello"); 5 end
```

However, the code

```
def f()
  3+5
  2
end
```

gives rise to the following warning

```
At line 2, char 2-4:
Warning 3: This expression should have type unit.
```

The reason is that this function is first computing the result of `3+5` and then returning 2 without doing anything with the result of the addition, and the fact that the type of `3+5` is not unit (it is `int`) allows to detect that. It is often the sign of a mistake when one computes something without using it; if however it is on purpose, you should use the `ignore` function to explicitly ignore the result:

```
def f()
  ignore(3+5)
  2
end
```

Lists. Some more elaborate values can be constructed by combining the previous ones. A first kind is *lists* which are finite sequences of values, being all of the same type. They are constructed by square bracketing the sequence whose elements are separated by commas. For instance, the list

```
[1, 4, 5]
```

is a list of three integers (1, 4 and 5), and its type is `[int]`, and the type of `["A", "B"]` would obviously be `[string]`. Note that a list can be empty: `[]`. The function `list.hd` returns the *head* of the list, that is its first element:

```
# list.hd([1, 4, 5]);;
- : int = 1
```

This function also takes an optional argument `default` which is the value which is returned on the empty list, which does not have a first element:

```
# list.hd(default=0, []);;
- : int = 0
```

Similarly, the `list.tl` function returns the *tail* of the list, i.e. the list without its first element (by convention, the tail of the empty list is the empty list). Other useful functions are

- `list.add`: add an element at the top of the list

  ```
  # list.add(5, [1, 3]);;
  - : [int] = [5, 1, 3]
  ```

- `list.length`: compute the length of a list

  ```
  # list.length([5, 1, 3]);;
  - : int = 3
  ```

- `list.mem`: check whether an element belongs to a list

  ```
  # list.mem(2, [1, 2, 3]);;
  - : bool = true
  ```

- `list.map`: apply a function to all the elements of a list

  ```
  # list.map(fun(n) -> 2*n, [1, 3, 5]);;
  - : [int] = [2, 6, 10]
  ```

- `list.iter`: execute a function on all the elements of a list

  ```
  # list.iter(fun(n) -> print(newline=false, n), [1, 3, 5]);;
  135- : unit = ()
  ```

- `list.nth`: return the n-th element of a list

  ```
  # list.nth([5, 1, 3], 2);;
  - : int = 3
  ```

  (note that the first element is the one at index n=0).

- `list.append`: construct a list by taking the elements of the first list and then those of the second list

  ```
  # list.append([1, 3], [2, 4, 5]);;
  - : [int] = [1, 3, 2, 4, 5]
  ```

Tuples. Another construction present in Liquidsoap is *tuples* of values, which are finite sequences of values which, contrarily to lists, might have different types. For instance,

```
(3, 4.2, "hello")
```

is a triple (a tuple with three elements) of type

```
int * float * string
```

which indicate that the first element is an integer, the second a float and the third a string. In particular, a *pair* is a tuple with two elements. For those, the first and second element can be retrieved with the functions `fst` and `snd`:

```
# p = (3, "a");;
p : int * string = (3, "a")
# fst(p);;
- : int = 3
# snd(p);;
- : string = "a"
```

For general tuples, there is a special syntax in order to access their elements. For instance, if `t` is the above tuple `(3, 4.2, "hello")`, we can write

```
let (n, x, s) = t
```

which will assign the first element to the variable `n`, the second element to the variable `x` and the third element to the variable `s`:

```
# t = (3, 4.2, "hello");;
t : int * float * string = (3, 4.2, "hello")
# let (n, x, s) = t;;
(n, x, s) : int * float * string = (3, 4.2, "hello")
# n;;
- : int = 3
# x;;
- : float = 4.2
# s;;
- : string = "hello"
```

Association lists. A quite useful combination of the two previous data structures is *association lists*, which are lists of pairs. Those can be thought of as some kind of dictionary: each pair is an entry whose first component is its key and second component is its value. These are the way metadata are represented for instance: they are lists of pairs of strings, the first string being the name of the metadata, and the second its value. For instance, a metadata would be the association list

```
m = [("artist", "Frank Sinatra"), ("title", "Fly me to the moon")]
```

indicating that the artist of the song is "Frank Sinatra" and the title is "Fly me to the moon". For such an association list, one can obtain the value associated to a given key using the `list.assoc` function:

```
list.assoc("title", m)
```

will return `"Fly me to the moon"`, i.e. the value associated to `"title"`. Since this is so useful, we have a special notation for the above function, and it is equivalent to write

```
m["title"]
```

to obtain the `"title"` metadata. Other useful functions are

- `list.assoc.mem`: determine whether there is an entry with a given key,
- `list.assoc.remove`: remove all entries with given key.

Apart from metadata, association lists are also used to store HTTP headers (e.g. in `http.get`).

In passing, you should note the importance of parenthesis when defining pairs. For instance

```
["a", "b"]
```

is a list of strings, whereas

```
[("a", "b")]
```

is a list of pairs of strings, i.e. an association list.

5.4 Programming primitives

Variables. We have already seen many examples of uses of *variables*: we use

```
x = e
```

in order to assign the result of evaluating an expression `e` to a variable `x`, which can later on be referred to as `x`. Variables can be masked: we can define two variables with the same name, and at any point in the program the last defined value for the variable is used:

```
n = 3
print(n)
n = n + 2
print(n)
```

will print `3` and `5`. Contrarily to most languages, the value for a variable cannot be changed (unless we explicitly require this by using references, see below), so the above program does not modify the value of `n`, it is simply that a new `n` is defined.

There is an alternative syntax for declaring variables which is

```
def x =
  e
end
```

It has the advantage that the expression `e` can spread over multiple lines and thus consist of multiple expressions, in which case the value of the last one will be assigned to `x`, see also next section. This is particularly useful to use local variables when defining a value. For instance, we can assign to `x` the square of sin(2) by

```
def x =
  y = sin(2.)
  y*y
end
```

Note that we first compute sin(2) in a variable `y` and then multiply `y` by itself, which avoids computing sin(2) twice. Also, the variable `y` is *local*: it is defined only until the next `end`, so that

```
y = 5
def x =
  y = sin(2.)
```

```
  y*y
end
print(y)
```

will print 5: outside the definition of x, the definition of y one on the first line is not affected by the local redefinition.

When we define a variable, it is generally to use its value: otherwise, why bothering defining it? For this reason, Liquidsoap issues a warning when an *unused* variable is found, since it is likely to be a bug. For instance, on

```
n = 2 + 2
```

Liquidsoap will output

```
Line 1, character 1:
Warning 4: Unused variable n
```

If this situation is really wanted, you should use `ignore` in order to fake a use of the variable `n` by writing

```
ignore(n)
```

Another possibility is to assign the special variable _, whose purpose is to store results which are not going to be used afterwards:

```
_ = 2 + 2
```

References. As indicated above, by default, the value of a variable cannot be changed. However, one can use a *reference* in order to be able to do this. Those can be seen as memory cells, containing values of a given fixed type, which can be modified during the execution of the program. They are created with the `ref` keyword, with the initial value of the cell as argument. For instance,

```
r = ref(5)
```

declares that `r` is a reference which contains `5` as initial value. Since `5` is an integer (of type `int`), the type of the reference `r` will be

```
ref(int)
```

meaning that its a memory cell containing integers. On such a reference, two operations are available:

- one can obtain the value of the reference by using the `!` keyword before the reference, so that `!r` denotes the value contained in the reference `r`, for instance

  ```
  x = !r + 4
  ```

  declares the variable `x` as being 9 (which is 5+4),

- one can change the value of the reference by using the `:=` keyword, e.g.

  ```
  r := 2
  ```

  will assign the value 2 to `r`.

The behavior of references can be illustrated by the following simple interactive session:

```
# r = ref(5);;
r : ref(int) = ref(5)
```

```
# !r;;
- : int = 5
# r := 2;;
- : unit = ()
# !r;;
- : int = 2
```

Note that the type of a reference is fixed: once `r` is declared to be a reference to an integer, as above, one can only put integers into it, so that the script

```
r = ref(5)
r := "hello"
```

will raise the error

```
Error 5: this value has type ref(int)
but it should be a subtype of ref(string)
```

which can be explained as follows. On the first line, the declaration `r = ref(5)` implies that `r` is of type `ref(int)` since it initially contains an integer. However, on the second line, we try to assign a string to `r`, which would only be possible if `r` was a reference to a string, i.e., of type `ref(string)`. Since `r` cannot have both types `ref(int)` and `ref(string)`, an error is raised.

Loops. The usual looping constructions are available in Liquidsoap. The `for` loop repeatedly executes a portion of code with an integer variable varying between two bounds, being increased by one each time. For instance, the following code will print the integers `1`, `2`, `3`, `4` and `5`, which are the values successively taken by the variable `i`:

```
for i = 1 to 5 do
  print(i)
end
```

In practice, such loops could be used to add a bunch of numbered files (e.g. `music1.mp3`, `music2.mp3`, `music3.mp3`, etc.) in a request queue for instance.

The `while` loop repeatedly executes a portion of code, as long a condition is satisfied. For instance, the following code doubles the contents of the reference `n` as long as its value is below `10`:

```
n = ref(1)
while !n < 10 do
  n := !n * 2
end
print(!n)
```

The variable `n` will thus successively take the values `1`, `2`, `4`, `8` and `16`, at which point the looping condition `!n < 10` is not satisfied anymore and the loop is exited. The printed value is thus `16`.

5.5 Functions

Liquidsoap is built around the notion of function: most operations are performed by those. For some reason, we sometimes call *operators* the functions acting on sources. Liquidsoap includes a standard library which consists of functions defined in the Liquidsoap language,

including fairly complex operators such as `playlist` which plays a playlist or `crossfade` which takes care of fading between songs.

Basics. A function is a construction which takes a bunch of arguments and produces a result. For instance, we can define a function `f` taking two float arguments, prints the first and returns the result of adding twice the first to the second:

```
def f(x, y)
  print(x)
  2*x+y
end
```

This function can also be written on one line if we use semicolons (`;`) to separate the instructions instead of changing line:

```
def f(x, y) = print(x); 2*x+y end
```

The type of this function is

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

The arrow `->` means that it is a function, on the left are the types of the arguments (here, two arguments of type `int`) and on the right is the type of the returned value of the function (here, `int`). In order to use this function, we have to apply it to arguments, as in

```
f (3, 4)
```

This will trigger the evaluation of the function, where the argument `x` (resp. `y`) is replaced by `3` (resp. `4`), i.e., it will print `3` and return the evaluation of `2*3+4`, which is `10`. Of course, generally, there is no reason why all arguments and the result should have the same type as in the above example, for instance:

```
# def f(s, x) = string.length(s) + int_of_float(x) end;;
f : (string, float) -> int = <fun>
```

As explained earlier, declarations of variables made inside the definition of a function are *local*: they are only valid within this definition (i.e., until the next `end`). For instance, in the definition

```
def f(x) =
  y = sin(x)
  y*y
end
```

the variable `y` is not available after the definition.

Handlers. A typical use of functions in Liquidsoap is for *handlers*, which are functions to be called when a particular event occurs, specifying the actions to be taken when it occurs. For instance, the `source.on_metadata` operator allows registering a handler when metadata occurs in a stream. Its type is

```
(source('a), (([string * string]) -> unit)) -> unit
```

and it thus takes two arguments:

- the source, of type `source('a)`, see below, whose metadata are to be watched,
- the handler, which is a function of type

`([string * string]) -> unit`

which takes as argument an association list (of type `[string * string]`) encoding the metadata and returns nothing meaningful (`unit`).

When some metadata occur in the source, the handler is called with the metadata as argument. For instance, we can print the title of every song being played on our radio (a source named `radio`) with

```
def handle_metadata(m) =
  print(m["title"])
end
source.on_metadata(radio, handle_metadata)
```

The handler is here the function `handle_metadata`, which prints the field associated to `"title"` in the association list given in the argument `m`.

Other useful operators allow the registration of handlers for the following situations:

- `blank.detect`: when a source is streaming blank (no sound has been streamed for some period of time),
- `source.on_track`: when a new track is played,
- `source.on_end`: when a track is about to end,
- `on_start` and `on_shutdown`: when Liquidsoap is starting or stopping.

Many other operators also take more specific handlers as arguments. For instance, the operator `input.harbor`, which allows users to connect to a Liquidsoap instance and send streams, has `on_connect` and `on_disconnect` arguments which allow the registration of handlers for the connection and disconnection of users.

Anonymous functions. For concision in scripts, it is possible define a function without giving it a name, using the syntax

```
fun (x) -> ...
```

This is called an *anonymous function*, and it is typically used in order to specify short handlers in arguments. For instance, the above example for printing the title in metadatas could equivalently be rewritten as

```
source.on_metadata(radio, fun (m) -> print(m["title"]))
```

where we define the function directly in the argument.

As a side note, this means that a definition of a function of the form

```
def f(x) =
  ...
end
```

could equivalently be written

```
f = fun (x) -> ...
```

When using this syntax, on the right hand of `->` Liquidsoap expects exactly one expression. If you intend to use multiple ones (for instance, in order to perform a sequence of actions), you can use the `begin ... end` syntax, which allows grouping multiple expressions as one. For instance,

```
handle_metadata = fun (m) -> begin
  print(m["artist"])
  print(m["title"])
end
source.on_metadata(radio, handle_metadata)
```

Anonymous function with no arguments. You will see that it is quite common to use anonymous functions with no arguments. For this reason, we have introduced a special convenient syntax for those and allow writing

```
{...}
```

instead of

```
fun () -> ...
```

Labeled arguments. A function can have an arbitrary number of arguments, and when there are many of them it becomes difficult to keep track of their order and their order matter! For instance, the following function computes the sample rate given a number of samples in a given period of time:

```
def samplerate(samples, duration) = samples / duration end
```

which is of type

```
(float, float) -> float
```

For instance, if you have 110250 samples over 2.5 seconds the samplerate will be `samplerate(110250., 2.5)` which is 44100. However, if you mix the order of the arguments and type `samplerate(2.5, 110250.)`, you will get quite a different result (2.27×10^{-5}) and this will not be detected by the typing system because both arguments have the same type. Fortunately, we can give *labels* to arguments in order to prevent this, which forces explicitly naming the arguments. This is indicated by prefixing the arguments with a tilde "~":

```
def samplerate(~samples, ~duration) = samples / duration end
```

The labels will be indicated as follows in the type:

```
(samples : float, duration : float) -> float
```

Namely, in the above type, we read that the argument labeled `samples` is a float and similarly for the one labeled `duration`. For those arguments, we have to give the name of the argument when calling the function:

```
samplerate(samples=110250., duration=2.5)
```

The nice byproduct is that the order of the arguments does not matter anymore, the following will give the same result:

```
samplerate(duration=2.5, samples=110250.)
```

Of course, a function can have both labeled and non-labeled arguments.

Optional arguments. Another useful feature is that we can give *default values* to arguments, which thus become *optional*: if, when calling the function, a value is not specified for such arguments, the default value will be used. For instance, if for some reason we tend to generally measure samples over a period of 2.5 seconds, we can make this become the value for the `duration` parameter:

```
def samplerate(~samples, ~duration=2.5) = samples / duration end
```

In this way, if we do not specify a value for the duration, its value will implicitly be assumed to be 2.5, so that the expression:

```
samplerate(samples=110250.)
```

will still evaluate to 44100. Of course, if we want to use another value for the duration, we can still specify it, in which case the default value will be ignored:

```
samplerate(samples=132300., duration=3.)
```

The presence of an optional argument is indicated in the type by prefixing the corresponding label with "?", so that the type of the above function is

```
(samples : float, ?duration : float) -> float
```

Actual examples. As a more concrete example of labeled arguments, we can see that the type of the operator `output.youtube.live`, which outputs a video stream to YouTube, is

```
(?id : string, ?video_bitrate : int, ?audio_encoder : string, ?audio_bitrate :
↪  int, ?url : string, key : string, source) -> source
```

(we have only slightly simplified the type `source`, which will only be detailed in a next section). Even if we have not read the documentation of this function, we can still guess what it is doing:

- there are 5 optional arguments that we should be able to ignore because they have reasonable default values (although we can guess the use of most of them from the label, e.g. `video_bitrate` should specify the bitrate we want to encode video, etc.),
- there is 1 mandatory argument which is labeled `key` of type `string`: it must be the secret key we need in order to broadcast on our YouTube account,
- there is 1 mandatory argument, unlabeled, of type `source`: this is clearly the source that we are going to broadcast to YouTube.

As we can see the types and labels of arguments already provide us with much information about the functions and prevent many mistakes.

If you want a more full-fledged example, have a look at the type of `output.icecast`:

```
(?id : string, ?chunked : bool, ?connection_timeout : float, ?description :
↪  string, ?dumpfile : string, ?encoding : string, ?fallible : bool, ?format :
↪  string, ?genre : string, ?headers : [string * string], ?host : string, ?icy_id
↪  : int, ?icy_metadata : string, ?mount : string, ?name : string, ?on_connect :
↪  (() -> unit), ?on_disconnect : (() -> unit), ?on_error : ((string) -> float),
↪  ?on_start : (() -> unit), ?on_stop : (() -> unit), ?password : string, ?port :
↪  int, ?protocol : string, ?public : bool, ?start : bool, ?timeout : float, ?url
↪  : string, ?user : string, ?verb : string, format('a), source) -> source
```

Although the function has 31 arguments, it is still usable because most of them are optional so that they are not usually specified. In passing, we recognize some of the concepts introduced earlier: the headers (`header` parameter) are coded as an association list, and there are quite few handlers (`on_connect`, `on_disconnect`, etc.).

Polymorphism. Some functions can operate on values of many possible types. For instance, the function `list.tl`, which returns the tail of the list (the list without its first element), works on lists of integers so that it can have the type

```
([int]) -> [int]
```

but it also works on lists of strings so that it can also have the type

```
([string]) -> [string]
```

and so on. In fact, this would work for any type, which is why in Liquidsoap the function `list.tl` is actually given the type

```
(['a]) -> ['a]
```

which means: "for whichever type you replace `'a` with, the resulting type is a valid type for the function". Such a function is called *polymorphic*, in the sense that it can be given multiple types: here, `'a` is not a type but rather a "meta-type" (the proper terminology is a *type variable*) which can be replaced by any regular type. Similarly, the empty list `[]` is of type `['a]`: it is a valid list of whatever type. More interestingly, the function `fst` which returns the first element of a pair has the type

```
('a * 'b) -> 'a
```

which means that it takes as argument a pair of a something (`'a`) and a something else (`'b`) and returns a something (`'a`). For instance, the type

```
(string * int) -> string
```

is valid for `fst`. In general, a type can involve an arbitrary number of type variables which are labeled `'a`, `'b`, `'c` and so on.

Constraints. In Liquidsoap, some type variables can also be constrained so that they cannot be replaced by any type, but only specific types. A typical example is the multiplication function `*`, which operates on both integers and floats, and can therefore be given both the types

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

and

```
(float, float) -> float
```

but not the type

```
(string, string) -> string
```

If you have a look at the type of `*` in Liquidsoap, it is

```
('a, 'a) -> 'a where 'a is a number type
```

which means that it has type `('a, 'a) -> 'a` where `'a` can only be replaced by a type that represents a number (i.e., `int` or `float`). Similarly, the comparison function `<=` has type

```
('a, 'a) -> bool where 'a is an orderable type
```

which means that it has the type `('a, 'a) -> bool` for any type `'a` on which there is a canonical order (which is the case of all usual types, excepting for function types and source types).

Getters. We often want to be able to dynamically modify some parameters in a script. For instance, consider the operator `amplify`, which takes a float and an audio source and returns the audio amplified by the given volume factor: we can expect its type to be

```
(float, source('a)) -> source('a)
```

so that we can use it to have a radio consisting of a microphone input amplified by a factor 1.2 by

```
mic   = input.alsa()
radio = amplify(1.2, mic)
```

In the above example, the volume 1.2 was supposedly chosen because the sound delivered by the microphone is not loud enough, but this loudness can vary from time to time, depending on the speaker for instance, and we would like to be able to dynamically update it. The problem with the current operator is that the volume is of type `float` and a float cannot change over time: it has a fixed value.

In order for the volume to have the possibility to vary over time, instead of having a `float` argument for `amplify`, we have decided to have instead an argument of type

```
() -> float
```

This is a function which takes no argument and returns a float (remember that a function can take an arbitrary number of arguments, which includes zero arguments). It is very close to a float excepting that each time it is called the returned value can change: we now have the possibility of having something like a float which varies over time. We like to call such a function a *float getter*, since it can be seen as some kind of object on which the only operation we can perform is get the value. For instance, we can define a float getter by

```
n = ref(0.)
def f ()
  n := !n + 1.
  !n
end
```

Each time we call `f`, by writing `f()` in our script, the resulting float will be increased by one compared to the previous one: if we try it in an interactive session, we obtain

```
# f();;
- : float = 1.0
# f();;
- : float = 2.0
# f();;
- : float = 3.0
```

Since defining such arguments often involves expressions of the form

```
fun () -> e
```

which is somewhat heavy, we have introduced the alternative syntax

```
{e}
```

for it.

Variations on a volume. The type of `amplify` is thus actually

```
(() -> float, source('a)) -> source('a)
```

and the operator will regularly call the volume function in order to have the current value for the volume before applying it. To be precise, it is actually called before each frame, which means roughly every 0.04 second. Let's see how we can use this in scripts. We can, of course, still apply a constant factor with

```
def volume () = 1.2 end
radio = amplify(volume, mic)
```

or, using anonymous functions,

```
radio = amplify(fun () -> 1.2, mic)
```

which we generally write, using the alternative syntax,

```
radio = amplify({1.2}, mic)
```

More interestingly, we can use the value of a float reference `v` for amplification:

```
radio = amplify({!v}, mic)
```

when the value of the reference gets changed, the amplification will get changed too.

In practice, float getters are often created using `interactive.float` which creates a float value which can be modified on the telnet server (this is an internal server provided by Liquidsoap on which other applications can connect to interact with it, as detailed in a later section), or `osc.float` which reads a float value from an external controller using the OSC library. For instance, with the script

```
volume = interactive.float("volume", 1.)
radio  = amplify(volume, mic)
```

the volume can be modified by issuing the telnet command

```
var.set volume = 0.5
```

You should remember that getters are regular functions. For instance, if we expect that the volume on telnet to be expressed in decibels, we can convert it to an actual amplification coefficient as follows:

```
volume = interactive.float("volume", 1.)
radio  = amplify({lin_of_dB(volume())}, mic)
```

As a more elaborate variation on this, let's program a fade in: the volume progressively increases from 0 to 1 in `fade_duration` seconds (here, 5 seconds). We recall that the volume function will be called before each frame, which is a buffer whose duration is called here `frame_duration` and can be obtained by querying the appropriate configuration parameter: in order to have the volume raise from 0 to 1, we should increase it by `frame_duration / fade_duration` at each call. If you execute the following script, you should thus hear a sine which is getting louder and louder during the 5 first seconds:

```
fade_duration  = 5.
frame_duration = get(default=1., "frame.duration")
v = ref(0.)

def volume ()
  v := !v + frame_duration / fade_duration
  if !v > 1. then v := 1. end
  log.important("Volume is now #{!v}")
  !v
end

s = amplify(volume, sine())
output(s)
```

Of course, this is for educational purposes only, and the actual way one would usually perform a fade in Liquidsoap is detailed in an ulterior section.

Let us give another advanced example, which uses many of the above constructions. The standard library defines a function `metadata.getter.float`, whose type is

```
(float, string, source('a)) -> source('a) * (() -> float)
```

which creates a float getter with given initial value (the first argument), which can be updated by reading a given metadata (the second argument) on a given source (the third argument). Its code is

```
def metadata.getter.float(init, metadata, s)
  x = ref(init)
  def f(m)
    s = m[metadata]
    if s != "" then x := float_of_string(s) end
  end
  source.on_metadata(s, f)
  {!x}
end
```

You can see that it create a reference `x`, which contains the current value, and registers a handler for metadata, which updates the value when the metadata is present, i.e. `m[metadata]` is different from the empty string `""`, which is the default value. Given a `radio` source which contains metadata labeled "`liq_amplify`", we can actually change the volume of the source according to the metadata with

```
volume = metadata.getter.float(1., "liq_amplify", radio)
radio  = amplify(volume, radio)
```

Constant or function. Finally, in order to simplify things a bit, you will see that the type of amplify is actually

```
({float}, source('a)) -> source('a)
```

where the type `{float}` means that both `float` and `() -> float` are accepted, so that you can still write constant floats where float getters are expected. What we actually call a *getter* is generally an element of such a type, which is either a constant or a function with no argument.

In order to work with such types, the standard library often uses the following functions:

- `getter`, of type `({'a}) -> {'a}`, creates a getter,
- `getter.get`, of type `({'a}) -> 'a`, retrieves the current value of a getter,
- `getter.function`, of type `({'a}) -> () -> 'a`, creates a function from a getter.

Recursive functions. Liquidsoap supports functions which are *recursive*, i.e., that can call themselves. For instance, in mathematics, the factorial function on natural numbers is defined as fact(n)=1×2×3×…×n, but it can also be defined recursively as the function such that fact(0)=1 and fact(n)=n×fact(n-1) when n>0: you can easily check by hand that the two functions agree on small values of n (and prove that they agree on all values of n). This last formulation has the advantage of immediately translating to the following implementation of factorial:

```
def rec fact(n) =
  if n == 0 then 1
  else n * fact(n-1) end
end
```

for which you can check that `fact(5)` gives 120, the expected result. As another example, the `list.length` function, which computes the length of a list, can be programmed in the following way in Liquidsoap:

```
def rec length(l)
  if l == [] then 0
  else 1 + length(list.tl(l)) end
end
```

We do not detail much further this trait since it is unlikely to be used for radios, but you can see a few occurrences of it in the standard library.

Partial evaluation. The final thing to know about functions in Liquidsoap is that they support *partial evaluation* of functions. This means that if you call a function, but do not provide all the arguments, it will return a new function expecting only the remaining arguments. For instance, consider the multiplication function

```
def mul(x, y) = x * y end
```

which is of type

```
(float, float) -> float
```

taking two floats and returning their products. We can then define a function which will compute the double of its input by

```
double = mul(2.)
```

which is of type

```
(float) -> float
```

Since we have provided only the first argument to `mul`, the `double` will define is still a function waiting for a second argument `x` and returning `mul(2., x)`, as we can see in the interactive mode:

```
# def mul(x, y) = x * y end;;
mul : (float, float) -> float = <fun>
# double = mul(2.);;
double : (float) -> float = <fun>
# double(5.);;
- : float = 10.0
```

A typical use of this is when providing arguments which are functions. For instance, if we want to print all the elements of a list without new lines between them, we can do

```
list.iter(print(newline=false), [1, 2, 3])
```

Here, the function `print` is of type

```
(?newline : bool, 'a) -> unit
```

and we only provide one argument (the one labeled `newline`) out of two. Without partial evaluation, we would have had to write

```
list.iter(fun (x) -> print(newline=false, x), [1, 2, 3])
```

which is somewhat more heavy.

5.6 Records and modules

Records. Suppose that we want to store and manipulate structured data. For instance, a list of songs together with their duration and tempo. One way to store each song is as a tuple of type `string * float * float`, but there is a risk of confusion between the duration and the length which are both floats, and the situation would of course be worse if there were more fields. In order to overcome this, one can use a *record* which is basically the same as a tuple, excepting that fields are named. In our case, we can store a song as

```
song = { filename = "song.mp3", duration = 257., bpm = 132. }
```

which is a record with three fields respectively named `filename`, `duration` and `bpm`. The type of such a record is

```
{filename : string, duration : float, bpm : float}
```

which indicates the fields and their respective type. In order to access a field of a record, we can use the syntax `record.field`. For instance, we can print the duration with

```
print("The duration of the song is #{song.duration} seconds")
```

Modules. Records are heavily used in Liquidsoap in order to structure the functions of the standard library. We tend to call *module* a record with only functions, but this is really the same as a record. For instance, all the functions related to lists are in the `list` module and functions such as `list.hd` are fields of this record. For this reason, the `def` construction allows adding fields in record. For instance, the definition

```
def list.last(l)
  list.nth(l, list.length(l)-1)
end
```

adds, in the module `list`, a new field named `last`, which is a function which computes the last element of a list. Another shorter syntax to perform definitions consists in using the `let` keyword which allows assigning a value to a field, so that the previous example can be rewritten as

```
let list.last = fun(l) -> list.nth(l, list.length(l)-1)
```

If you often use the functions of a specific module, the `open` keyword allows using its fields without having to prefix them by the module name. For instance, in the following example

```
open list
x = nth(l, length(l)-1)
```

the `open list` directive allows directly using the functions in this module: we can simply write `nth` and `length` instead of `list.nth` and `list.length`.

Values with fields. A unique feature of the Liquidsoap language is that it allows adding fields to any value. We also call them *methods* by analogy with object-oriented programming. For instance, we can write

```
song = "test.mp3".{duration = 123., bpm = 120.}
```

which defines a string (`"test.mp3"`) with two methods (`duration` and `bpm`). This value has type

```
string.{duration : float, bpm : float}
```

and behaves like a string, e.g. we can concatenate it with other strings:

```
print("the song is " ^ song)
```

but we can also invoke its methods like a record or a module:

```
print("the duration is #{song.duration}")
```

The construction **`def replaces`** allows changing the main value while keeping the methods unchanged, so that

```
def replaces song = "newfile.mp3" end
print(song)
```

will print

```
"newfile.mp3".{duration = 123., bpm = 120.}
```

(note that the string is modified but not the fields `duration` and `bpm`).

Examples. The `http.get` function, which retrieves a webpage over HTTP, has the type:

```
(?headers : [string * string], ?timeout : float, string) ->
string.{headers : [string * string],
        status_message : string,
        status_code : int,
        protocol_version : string}
```

It returns a string (the contents of the webpage) with fields specifying the returned headers, the status message and the version used by the protocol. A typical use is

```
h = http.get("http://www.google.fr/xxx")
if h.status_code < 400 then
  print("Contents of the webpage: #{h}")
else
  print("An error occured: #{h.status_code} (#{h.status_message})")
end
```

Another typical example is the `rms` operator, which takes a source as argument, and returns the same source together with an added method named `rms` which allows retrieving the current value for the RMS (which is a measure of sound intensity). The RMS of a source can thus be logged every second in a file as follows (functions concerning files and threads are explained in there):

```
s = playlist("~/Music")
s = rms(s)
def save_metrics() = file.write(data="RMS: #{s.rms()}", "/tmp/rms") end
thread.run(every=1., save_metrics)
output(s)
```

When the return type of a function has methods, the help of Liquidsoap displays them in a dedicated section. For instance, every function returning a source, also returns methods associated to this source, such as the `skip` function which allows skipping the current track

(those methods are detailed in a section below). If we ask for help about the `playlist` operator by typing

```
$ liquidsoap -h playlist
```

we can observe this: the help displays, among other,

```
Methods:

 * reload : (?uri : string) -> unit
     Reload the playlist.

 * skip : () -> unit
     Skip to the next track.
```

This indicates that the returned source has a `reload` method, which allows reloading the playlist, possibly specifying a new file, as well as the `skip` method described above. If you try at home, you will see that they are actually many more methods.

5.7 Advanced values

In this section, we detail some more advanced values than the ones presented in previous sections. You are not expected to be understanding those in details for basic uses of Liquidsoap.

Errors. In the case where a function does not have a sensible result to return, it can raise an *error*. Typically, if we try to take the head of the empty list without specifying a default value (with the optional parameter `default`), an error will be raised. By default, this error will stop the script, which is usually not a desirable behavior. For instance, if you try to run a script containing

```
list.hd([])
```

the program will exit printing

```
Error 14: Uncaught runtime error:
type: not_found, message: "no default value for list.hd"
```

This means that the error named "`not_found`" was raised, with a message explaining that the function did not have a reasonable default value of the head to provide.

In order to avoid this, one can *catch* exceptions with the syntax

```
try
  code
catch err do
  handler
end
```

This will execute the instructions `code`: if an error is raised at some point during this, the code `handler` is executed, with `err` being the error. For instance, instead of writing

```
l = []
x = list.hd(default=0, l)
```

we could equivalently write

```
l = []
x =
  try
    list.hd(l)
  catch err do
    0
  end
```

The name and message associated to an error can respectively be retrieved using the functions `error.kind` and `error.message`, e.g. we can write

```
try
  ...
catch err do
  print("the error #{error.kind(err)} was raised")
  print("the error message is #{error.message(err)}")
end
```

Typically, when reading from or writing to a file, errors will be raised when a problem occurs (such as reading from a non-existent file or writing a file in a non-existent directory) and one should always check for those and log the corresponding message:

```
try
  file.write(data=data, "/non/existent/path")
catch err do
  log.important("Could not write to file: #{error.message(err)}")
end
```

Specific errors can be catched with the syntax

```
try
  ...
catch err in l do
  ...
end
```

where `l` is a list of error names that we want to handle here.

Errors can be raised from Liquidsoap with the function `error.raise`, which takes as arguments the error to raise and the error message. For instance:

```
error.raise(error.not_found, "we could not find your result")
```

Finally, we should mention that all the errors should be declared in advance with the function `error.register`, which takes as argument the name of the new error to register:

```
myerr = error.register("my_error")
error.raise(myerr, "testing my own error")
```

Nullable values. It is sometimes useful to have a default value for a type. In Liquidsoap, there is a special value for this, which is called `null`. Given a type `t`, we write `t?` for the type of values which can be either of type `t` or be `null`: such a value is said to be *nullable*. For instance, we could redefine the `list.hd` function in order to return null (instead of raising an error) when the list is empty:

```
def list.hd(l)
  if l == [] then null() else list.hd(l) end
```

```
end
```

whose type would be

```
([a]) -> a?
```

since it takes as argument a list whose elements are of type `'a` and returns a list whose elements are `'a` or `null`. As it can be observed above, the null value is created with `null()`.

In order to use a nullable value, one typically uses the construction `x ?? d` which is the value `x` excepting when it is null, in which case it is the default value `d`. For instance, with the above head function:

```
x = list.hd(l)
print("the head is " ^ (x ?? "not defined"))
```

Some other useful functions include

- `null.defined`: test whether a value is null or not,
- `null.get`: obtain the value of a nullable value supposed to be distinct from `null`,
- `null.case`: execute a function or another, depending on whether a value is null or not.

5.8 Configuration and preprocessor

Liquidsoap has a number of features (such as its preprocessor) which allow useful operations on the scripts, but cannot really be considered as part of the core language itself. Those are presented below.

Configuration. The main configuration options can be set by using the `set` function, which takes as arguments the name of the name of the setting (a string) and the value for this setting, whose type depends on the setting. These settings affect the overall behavior of Liquidsoap. For instance, we can have Liquidsoap use a 48kHz samplerate for audio (the default being 44.1kHz) by adding the following command at the beginning of our script:

```
set("frame.audio.samplerate", 48000)
```

or we can increase the verbosity of the log messages with

```
set("log.level", 4)
```

which sets the maximum level of shown log messages to 4, the default being 3. We recall that the log levels are 1 for critical messages, 2 for severe issues, 3 for important messages, 4 for information and 5 for debug messages.

Dually, we can obtain the value of an argument with the `get` function, e.g.

```
print('Samplerate is #{get(default=0, "frame.audio.samplerate")}.')
```

As you can see, in addition to the name of the setting, this function takes a parameter labeled `default` which is the value which is to be returned if the setting does not exist. You can obtain the list of all available settings, as well as their default value with the command

```
liquidsoap --conf-descr
```

and help on a particular setting can be obtained with `--conf-descr-key`: for instance,

```
liquidsoap --conf-descr-key frame.duration
```

will print

```
# Tentative frame duration in seconds

Audio samplerate and video frame rate constrain the possible frame
durations. This setting is used as a hint for the duration, when
`frame.audio.size` is not provided.

    set("frame.duration", 0.04)
```

The value `0.04` at the bottom indicates the default value.

Including other files. It is often useful to split your script over multiple files, either because it has become quite large, or because you want to be able to reuse common functions between different scripts. You can include a file `file.liq` in a script by writing

```
%include "file.liq"
```

which will be evaluated as if you had pasted the contents of the file in place of the command.

For instance, this is useful in order to store passwords out of the main file, in order to avoid risking leaking those when handing the script to some other people. Typically, one would have a file `passwords.liq` defining the passwords in variables, e.g.

```
radio_pass = "secretpassword"
```

and would then use it by including it:

```
%include "passwords.liq"

radio = ...
output.icecast(%mp3, host="localhost", port=8000,
               password=radio_pass, mount="my-radio.mp3", radio)
```

so that passwords are not shown in the main script.

Conditional execution. Liquidsoap embeds a preprocessor which allows including or not part of the code depending on some conditions. For instance, the following script will print something depending on whether the function `input.alsa` is defined or not:

```
%ifdef output.alsa
print("We have support for ALSA.")
%else
print("We don't have support for ALSA.")
%endif
```

This is useful in order to have some code being executed depending on the compilation options of Liquidsoap (the above code will be run only when Liquidsoap has the support for the ALSA library) and is used intensively in the standard library. The command `%ifndef` can similarly be used to execute code when a function is not defined. We can also execute a portion of code whenever an encoder is present using `%ifencoder` (or `%ifnencoder` when an encoder is not present), the end of the code in question being delimited with `%endif` as above. For instance, suppose that we want to encode a file in MP3, if Liquidsoap was compiled with support for it, and otherwise default to wave. This can be achieved with

```
%ifencoder %mp3
output.file(%mp3, "out.mp3", s)
%endif
```

```
%ifnencoder %mp3
output.file(%wav, "out.wav", s)
%endif
```

Finally, the command `%ifversion` can be used to execute some code conditionally, depending on the version of Liquidsoap:

```
%ifversion >= 2.1
print("This is a very new version!")
%endif
```

This is quite useful in order to provide a script which is compatible with multiple versions of Liquidsoap (note that this functionality was only introduced in version 2.0, and thus unfortunately cannot be used in order to ensure backward compatibility with versions earlier than this).

5.9 Standard functions

In this section, we detail some of the most useful general purpose functions present in the standard library. The functions related to sound and streaming are mentioned in next section and detailed in subsequent chapters.

Type conversion. The string representation of any value can be obtained with the `string_of` function:

```
print(string_of([1,2,3]))
```

Most expected type conversion function are implemented with names of the form `A_of_B`. For instance, we can convert a string to an integer with `int_of_string`:

```
print(1 + int_of_string("2"))
```

Files. The whole contents of a file can be obtained with the function `file.contents`:

```
f = file.contents("test.txt")
print("The contents of the file is: " ^ f)
```

In the case where the file is big, it is advisable to use `file.read`, whose type is

```
(string) -> () -> string
```

and returns a function which successively reads chunks of the file until the end, in which case the empty string is returned. The contents of a file can be dumped using it by

```
f = file.read("test.txt")
r = ref(f())
while !r != "" do
  print(newline=false, !r)
  r := f()
end
```

Other useful functions are

- `file.exists`: test whether a file exists,
- `file.write`: write in a file,
- `file.remove`: remove a file,
- `file.ls`: list the files present in a directory.

Also, convenient functions for working on paths are present in the `file` and `path` module:

- `file.extension`: get the extension of a file,
- `file.temp`: generate a fresh temporary filename,
- `path.dirname`: get the directory of a path,
- `path.basename`: get the file name without the directory from a path,
- `path.home`: home directory of user,

and so on.

HTTP. Distant files can be retrieved over `HTTP` using `http.get`. For instance, the following script will fetch and display the list of changes in Liquidsoap:

```
c = http.get(
      "https://raw.github.com/savonet/liquidsoap/master/CHANGES.md")
print("Here are the latest changes in Liquidsoap:\n\n" ^ c)
```

Other useful functions are

- `http.post`: to send data, typically on forms,
- `http.put`: to upload data,
- `http.delete`: to delete resources.

Liquidsoap also features an internal web server called *harbor*, which allows serving web pages directly from Liquidsoap, which can be handy to present some data related to your script or implement some form of advanced interaction. This is described on details in there.

System. The arguments passed on the command line to the current script can be retrieved using the `argv` function. Its use is illustrated in there.

The current script can be stopped using the `shutdown` function which cleanly stops all the sources, and so on. In case of emergency, the application can be immediately stopped with the `exit` function, which allows specifying an exit code (the convention is that a non-zero code means that an error occurred). The current script can also be restarted using `restart`.

In order to execute other programs from Liquidsoap, you can use the function `process.read` which executes a command and returns the text it wrote in the standard output. For instance, in the script

```
n = process.read("find ~/Music -type f | wc -l")
n = int_of_string(string.trim(n))
print("We have #{n} files in the library.")
```

we use the `find` command to find files in the `~/Music` directory and pipe it through `wc -l` which will count the number of printed lines, and thus the number of files. In passing, in practice you would do this in pure Liquidsoap with

```
n = list.length(file.ls(recursive=true, "~/Music"))
```

There is also the quite useful variant called `process.read.lines`, which returns the list of lines written on the standard output. Typically, suppose that we have a script `generate-playlist` which outputs a list of files to play, one per line. We can play it by feeding it to `playlist.list` which plays a list of files:

```
p = process.read.lines("./generate-playlist")
s = playlist.list(p)
output(s)
```

The more elaborate variant `process.run` allows retrieving the return code of the program, set a maximal time for the execution of the program and *sandbox* its execution, i.e. restrict the directories it has access to in order to improve security (remember that executing programs is dangerous, especially if some user-contributed data is used). This is further detailed in there.

Threads. The function `thread.run` can be used to run a function asynchronously in a *thread*, meaning that the function will be executed in parallel to the main program and will not block other computations if it takes time. It takes two optional arguments:

- `delay`: if specified, the function will not be run immediately, but after the specified number of seconds,
- `every`: if specified, the function will be run regularly, every given number of seconds.

Phone ring. For instance, we can simulate the sound of a hanged phone by playing a sine and switching the volume on and off every second. This is easily achieved as follows:

```
volume = ref(0.)

def change()
  print("Changing state.")
  volume := 1. - !volume
end
thread.run(every=1., change)

s = amplify({!volume}, sine())
output(s)
```

Here, we amplify the sine by the contents of a reference `volume` (or, more precisely, by a getter which returns the value of the reference). Its value is switched between `0.` and `1.` every second by the function `change`.

Auto-gain control. A perhaps more useful variant of this is *auto-gain control*. We want to adjust the volume so that the output volume is always roughly -14 LUFS, which is a standard sound loudness measure. One way to do this is to regularly check its value and increase or lower the volume depending on whether we are below or above the threshold:

```
pre    = lufs(playlist("~/Music"))
volume = ref(1.)
post   = lufs(amplify({!volume}, pre))
def adjust()
  volume := !volume * lin_of_dB((-14. - post.lufs_momentary()) / 20.)
  volume := max(0.01, min(10., !volume))
  print(newline=false,
    "LUFS: #{pre.lufs()} -> #{post.lufs()} (volume: #{!volume})\r")
end
thread.run(adjust, every=0.1)
output(post)
```

Here, we have a source `pre` which we amplify by the value of the reference `volume` in order to define a source `post`. On both sources, the `lufs` function instructs that we should measure the LUFS, which value can be obtained by calling the `lufs` and `lufs_momentary` methods attached to the sources. Regularly (10 times per second), we run the function `adjust` which multiplies

the volume by the coefficient needed to reach -14 LUFS (to be precise, we actually divide the distance to -14 by 20 in order not to change the volume too abruptly, and we constrain the volume in the interval [0.01,10] in order to keep sane values).

Of course, in practice, you do not need to implement this by hand: the operator `normalize` does this for you, and more efficiently than in the above example. But it is nice to see that you could if you needed, to experiment with new strategies for managing the gain for instance.

Conditional execution. Another useful function is `thread.when`, which executes a function when a predicate (a boolean getter, of type `{bool}`) becomes true. By default, the value of the predicate is checked every second, this can be changed with the `every` parameter. For instance, suppose that we have a file named "song" containing the path to a song, and we want that each time we change the contents of this file, the new song is played. This can be achieved as follows:

```
q    = request.queue()
song = {file.contents("song")}
thread.when(getter.changes(song), {q.push(request.create(song()))})
output(q)
```

We begin by creating `q` which is a request queue, i.e. some source on which we can push new songs (those are detailed in there) and `song` which is a getter which returns the contents of the file. We then use `thread.when` on the predicate `getter.changes(song)` (which is true when the contents of `song` changes) in order to detect changes in `song` and, when this is the case, actually push the song on the request queue.

Time. In case you need it, the current time can be retrieved using the `time` function. This function returns the number of seconds since the 1st of January 1970, which is mostly useful to measure duration by considering the difference between two points in time. In order to convert this into more usual time notations you can use the functions `time.local` and `time.utc` which extract the usual information (year, month, day, hour, etc.), respectively according to the current time zone and the Greenwich median time, and return those in a record. For instance, we can print the current date with

```
t = time.local()
print("The current date is #{t.year+1900}-#{t.mon}-#{t.mday}.")
```

As a useful variant, the function `time.up` returns the *uptime* of the script, i.e. the number of seconds since its execution has begun.

5.10 Streams in Liquidsoap

Apart from the general-purpose constructions of the language described above, Liquidsoap also has constructions dedicated to building streams: after all this is what we are all here for. Those are put to practice in the next chapter and described in details in the chapter after. We however quickly recap here the main concepts and operators.

Sources. An operator producing a stream is called a *source* and has a type of the form

```
source(audio=..., video=..., midi=...)
```

where the "..." indicate the *contents* that the source can generate, i.e. the number of channels, and their nature, for audio, video and MIDI data, that the source can generate. For instance, the `playlist` operator has (simplified) type

```
(?id : string, string) -> source(audio='a, video='b, midi='c)
```

we see that it takes as parameters an optional string labeled `id` (most operators take such an argument which indicates its name, and is used in the logs or the telnet) as well as a string (the playlist to play) and returns a source (which plays the playlist...).

Some sources are *fallible*, which means that they are not always available. For instance, the sound input from a DJ over the internet is only available when the DJ connects. We recall from there that a source can be made infallible with the `mksafe` operator or by using a fallback to an infallible source.

Encoders. Some outputs need to send data encoded in some particular format. For instance, the operator which records a stream into a file, `output.file`, needs to know in which format we want to store the file in, such MP3, AAC, etc. This is specified by passing special parameters called *encoders*. For instance, the (simplified) type of `output.file` is

```
(?id : string, format('a), string, source('a)) -> unit
```

We see that it takes the `id` parameter (a string identifying the operator), an encoder (the type of encoders is `format(...)`), a string (the file where we should save data) and a source. This means that we can play our playlist and record it into an MP3 file as follows:

```
s = mksafe(playlist("~/Music"))
output.file(%mp3, "out.mp3", s)
```

Here, `%mp3` is an encoder specifying that we want to encode into the MP3 formats. Encoders for most usual formats are available (`%wav` for WAV, `%fdkaac` for AAC, `%opus` for Opus, etc.) and are detailed later on.

Requests. Internally, Liquidsoap does not directly deal with a file, but rather with an abstraction of it called a *request*. The reason is that some files require some processing before being accessible. For instance, we cannot directly access a distant MP3 file: we first need to download it and make sure that it has the right format.

This is the reason why most low-level operators do not take files as arguments, but requests. The main thing you need to know in practice is that you can create a request from a file location, using the `request.create` function. For instance, in the following example, we create a request queue `q`, on which we can add requests to play in it using `q.push`. We define a function `play`, which adds the file on the queue, by first creating a request from it. We then use `list.iter` to apply this function `play` on all the MP3 files of the current directory. The following script will thus play all the MP3 files in the current directory:

```
q = request.queue()
def play(file)
  r = request.create(file)
  q.push(r)
end
list.iter(play, file.ls(pattern="*.mp3", "."))
output(q)
```

Main functions. The main functions in order to create and manipulate audio streams are

- `playlist`: plays a playlist,
- `fallback`: plays the first available source in a list,
- `switch`: plays a source depending on a condition,

- `crossfade`: fade successive tracks,
- `output.icecast`, `output.hls`, `output.file`: output on Icecast, an HLS playlist, or in a file,
- `request.queue`: create a queue that can be dynamically be fed with user's requests and will play them in the order they were received.

Their use is detailed in next chapter.

6 Full workflow of a radio station

This chapter explains in details the main tools and techniques in order to setup a webradio. It essentially follows the introductory chapter, but gives much more details about techniques and parameters one can use to achieve his goals.

6.1 Inputs

Playlists. A radio generally starts with a *playlist*, which is simply a file containing a list of files to be played. The `playlist` operator does that: it takes as a playlist as argument and sequentially plays the files it contains. For instance, the script

```
s = playlist("my_playlist")
output(s)
```

will play all the files listed in the `my_playlist` playlist. The operator also accepts a directory as argument, in which case the playlist will consist of all the files in the directory: the script

```
s = playlist("~/Music")
output(s)
```

will play all the files in the `Music` directory. The format of a playlist generally consists in a list of files, with one file per line, such as

```
/data/mp3/file1.mp3
/data/mp3/file2.mp3
/data/mp3/file3.mp3
http://server/file.mp3
ftp://otherserver/file.mp3
```

but other more advanced playlist formats are also supported: pls, m3u, asx, smil, xspf, RSS podcasts, etc. Those are generally created by using dedicated software.

Playlist arguments. By default, the files are played in a random order but this can be changed with the `mode` parameter of `playlist` which can either be

- `"normal"`: play files in the order indicated in the playlist,
- `"randomize"`: play files in a random order chosen for the whole playlist at each round (default mode),

- `"random"`: pick a random file each time in the playlist (there could thus be repetitions in files).

In the first two modes, the `loop` argument indicates, when we have played all the files, whether we should start playing them again or not. By default, the playlist is never reloaded, i.e. changes brought to it are not taken in account, but this can be modified with the parameters `reload_mode` (which indicates when we should reload the playlist) and `reload` (which indicates how often we should reload the playlist). For instance:

- reload the playlist every hour (1 hour being 3600 seconds):

  ```
  s = playlist(reload=3600, reload_mode="seconds", "playlist")
  ```

- reload the playlist after each round (when the whole playlist has been played):

  ```
  s = playlist(reload=1, reload_mode="rounds", "playlist")
  ```

- reload the playlist whenever it is modified:

  ```
  s = playlist(reload_mode="watch", "playlist")
  ```

Playlists can also be reloaded from within the scripts by calling the `reload` method of a source produced by the `playlist` operator. For instance, reloading every hour can also be performed with

```
s = playlist("playlist")
thread.run(every=3600., {s.reload()})
```

The `reload` method take an optional argument labeled `uri` in case you want to specify a new playlist to load.

Another useful option is `check_next`, to specify a function which will determine whether a file should be played or not in the playlist: this function takes a request as argument and returns a boolean. For instance, we can ensure that only the files whose name end in ".MP3" are played with

```
def check(r)
  fname = request.uri(r)
  string.contains(suffix=".mp3", fname)
end
s = playlist(check_next=check, "~/Music")
```

The function `check` takes the request `r` given as argument, extracts its URI, and then returns true or false depending on whether this URI ends with ".MP3" or not. As another example, we can base our decision on the metadata of the file as follows:

```
def check(r)
  m = request.metadata(r)
  m["genre"] == "Rock"
end
s = playlist(check_next=check, "~/Music")
```

Here, we obtain the metadata with `request.metadata`, and declare that we should play a file only if its genre is "Rock" (remember that the metadata are encoded as an association list, as explained in there).

Playing lists of files. One inconvenient of the `check` function is that it is called whenever the `playlist` operator needs to prepare a new song, presumably because the current one

is about to end. This means that if too many songs are rejected in a row, we might fail to produce a valid next file in time. Another approach could consist in filtering the files we want at startup: this takes longer at the beginning, but it is more predictable and efficient on the long run. This approach is taken in the following script to play dance music songs:

```
def check(f)
  m = file.metadata(f)
  if m["genre"] == "Dance" then
    print("Keeping #{f}.")
    true
  else
    false
  end
end
l = playlist.files("~/Music/misc")
l = list.filter(check, l)
s = playlist.list(l)
```

We use `playlist.files` to obtain the list of files contained in our playlist of interest, then we use `list.filter` to only keep in this list the files which are validated by the function `check`, and finally we use the `playlist.list` operator to play the resulting list of files (`playlist.list` is a variant of `playlist` which takes a list of files to play instead of a playlist). The `check` function uses `file.metadata` in order to obtain the metadata for each file `f`: this can take quite some time if we have many files, but will be done only at the startup of the script.

Avoiding repetitions. A common problem with playlists in randomized order is that, when the playlist is reloaded, it might happen that a song which was played not so long ago is scheduled again. In order to avoid that, it is convenient to use the `playlog` operator which records when songs were played and can indicate when a song was last played. When called, it returns a record with two functions:

- `add` which records that a song with given metadata has just been played,
- `last` which returns how long ago a song with given metadata was last played, in seconds.

We can use this to reject, in a playlist, all the songs which have been played less than 1 hour (= 3600 seconds) ago as follows:

```
l = playlog()
def check(r)
  m = request.metadata(r)
  if l.last(m) < 3600. then
    log.info("Rejecting #{m['filename']} (played #{l.last(m)}s ago).")
    false
  else
    l.add(m)
    true
  end
end
s = playlist(check_next=check, "playlist")
output(s)
```

and thus avoid repetitions when reloading.

Interesting options of the `playlog` operator are:

- `duration` specifies the time after which the tracks are forgotten in the playlog (setting this avoids that it grows infinitely by constantly adding new tracks),
- `persistency` provides a file in which the list of songs is recorded, which allows preserving the playlog across restarts of the script,
- `hash` is a function which extracts the string identifying a file from its metadata: by default, we use the filename as identifier, but we could for instance use an MD5 checksum of the file by passing the function

  ```
  fun (m) -> file.digest(metadata.filename(m))
  ```

Single files. If you only need to play one file, you can avoid creating a playlist with this file only, by using the operator `single` which loops on one file. This operator is also more efficient in the case the file is distant because it is downloaded once for all:

```
s = single("http://server/file.mp3")
```

By the way, if you do not want to loop over and over the file, and only play it once, you can use the operator `once` which takes a source as argument, plays one song of this source, and becomes unavailable after that.

```
s = once(single("http://server/file.mp3"))
```

Distant streams. The operators `playlist` or `single` make sure in advance that the files to be played are available: in particular, they download distant files so that we are sure that they are ready when we need them. Because of this, they are not suitable in order to play continuous streams (which are very long, or can even be infinite), because Liquidsoap would try to download them entirely before reading them.

This is the reason why the `input.http` operator should be used instead in order to play a stream:

```
s = input.http("https://icecast.radiofrance.fr/fip-hifi.aac")
```

This operator works with streams such as those generated by Icecast, but also with playlists containing streams. It will regularly pull data from the given location, and therefore should be used for locations that are assumed to be available most of the time. If not, it might generate unnecessary traffic and pollute the logs: in this case, it is perhaps better to inverse the paradigm and use the `input.harbor` operator described below, which allows the distant stream to connect to Liquidsoap.

HLS streams. Streams in HLS format are quite different from the above ones: they consist of a rolling playlist of short audio segments, as explained in there. This is the reason why they are handled by a different operator, `input.hls`:

```
s = input.hls("https://stream.radiofrance.fr/fip/fip.m3u8")
```

Interactive playlists. Instead of having a static playlist, you might want to use you own script to generate the song which should be played next (e.g. you might fetch requests from users from the web or a database, or you might have a neural network deciding for you which song is the best to be played next). In order to proceed in this way, you should use the `request.dynamic` operator, which takes as argument a function returning the next song to be played: this function has type `() -> request('a)`, meaning that it takes no argument and returns a request. For instance, suppose that we have a script called `next-song`, which echoes

on the standard output the next song to be played on the standard output. A degenerate example of such a script, using the shell, could be

```
#!/bin/bash
echo "test.mp3"
```

which always returns `test.mp3` as song to be played, but of course you could use any program in any programming language as long as it outputs the file to be played on the standard output. We can then query this script in order to play song as follows:

```
def next()
  uri = list.hd(process.read.lines("./next-song"))
  request.create(uri)
end
s = request.dynamic(next)
```

Here, our `next` function executes the above script `next-song`, using the function `process.read.lines` which returns the list of lines returned by the script. We then take the first line with `list.hd` and create a request from from it using `request.create`. As a variant, suppose that the next song to be played is present in a file named `song`. We can play it as follows:

```
def next()
  uri = file.contents("song")
  if uri != "" then
    request.create(uri)
  else
    null()
  end
end
s = request.dynamic(retry_delay=1., next)
```

The `check` function now reads the contents of the file `song` and creates a request from it. In the case where the file is empty, there is no song to play, and we return the value `null` to indicate it. The `retry_delay` parameter of `request.dynamic` indicates that, in such an event, we should wait for 1 second before trying again. This example is not perfect: there is a chance that a given song will be played multiple times if we don't update the file `song` timely enough: we see a better way of achieving this kind of behavior in next section.

The playlist operator. We should mention here that our beloved `playlist` operator is actually implemented in Liquidsoap, in the standard library, using `request.dynamic`. Here is a simplified version of the definition of this function:

```
def playlist(~randomize=true, ~reload=true, p)
  l = ref(playlist.files(p))
  if randomize then l := list.shuffle(!l) end
  def next()
    if not (list.is_empty(!l)) then
      song = list.hd(!l)
      l := list.tl(!l)
      request.create(song)
    else
      if reload then l := playlist.files(p) end
      null()
    end
```

```
  end
  request.dynamic(next)
end
```

When creating the playlist, we first store the list of its files is a reference `l`, and randomize its order when the `randomize` parameter is true. We then use `request.dynamic` with a `next` function which returns a request made from the first element of the list (and this first element is removed from the list). When the `reload` parameter is true, we reload the list with the contents of the playlist when it becomes empty.

Request queues. In an interactive playlist, the operator asks for the next song. But in some situations, instead of this passive way of proceeding (you are asked for songs), you would rather have an active way of proceeding (you inform the operator of the new files to play when you have some). Typically, if you have a website where users can request songs, you would like to be able to put the requested song in a playlist at the moment the user requests it. This is precisely the role of the `request.queue` operator, which maintains a list of songs to be played in a *queue* (the songs are played in the order they are pushed). A typical setup involving this operator would be the following:

```
set("server.telnet", true)
playlist = playlist("~/Music")
queue    = request.queue()
radio    = fallback(track_sensitive=false, [queue, playlist])
output(radio)
```

We have both a playlist and a queue, and the radio is defined by using the `fallback` operator which tries to fetch the stream from the queue and defaults to the playlist if the queue is empty. The `track_sensitive=false` argument instructs that we should play the stream generated by the queue as soon as it is available: by default, `switch` will wait for the end of the current track before switching to the queue.

Pushing songs in a queue. You might wonder then: how do we add new songs in the queue? The role of the first line is to instruct Liquidsoap to start a "telnet" server, which is listening by default on port 1234, on which commands can be sent. The queue will register a new command on this server, so that if you connect to it and write `queue.push` followed by an URI, it will be pushed into the queue where it will wait for its turn to be played. In practice this can be done with commands such as

```
echo "queue.push test.mp3" | telnet localhost 1234
```

which uses the standard Unix tool `telnet` to connect to the server supposed to be running on the local host and listening on port 1234, and write the command "`queue.push test.mp3`" on this server, to which it will react by adding the song "`test.mp3`" in the queue. We refer the reader to this section for more details about the telnet server.

If you have multiple queues in your script, you give them names by specifying the `id` parameter of `request.queue`, which can be any string you want. In this case, the pushing command will be `ID.push` (where `ID` should be replaced by the actual identifier you specified), which clearly indicates in which queue you want to push. For instance, in the script

```
set("server.telnet", true)
playlist = playlist("~/Music")
queue1   = request.queue(id="q1")
queue2   = request.queue(id="q2")
```

```
radio    = fallback([queue1, queue2, playlist])
output(radio)
```

the two queues are respectively called `q1` and `q2`, so that we can push a song on the second queue by issuing the telnet command "`q2.push file.mp3`".

It is also possible to push a request into a queue directly from Liquidsoap by using the method `push` of a source defined by `request.queue`, or the method `push.uri` to push an URI. For instance, consider the following script

```
playlist = playlist("~/Music")
q        = request.queue()
radio    = add([q, playlist])
thread.run(every=60., {q.push.uri("say:Another minute has passed!")})
output(radio)
```

It sets up an auxiliary queue `q`, and uses the function `thread.run` to execute every minute a function which pushes in to the queue the URI `"say:Another minute has passed!"`. Because it begins by "`say:`" Liquidsoap will use a speech synthesis software to turn the text into audio, and we will hear "Another minute has passed" every minute, over the playlist (the `add` operator plays simultaneously all the sources in its input list).

Implementation of queues. Incidentally, the function `request.queue` is implemented in Liquidsoap, by using a list to maintain the queue of requests. Here is a slightly simplified version of it:

```
def request.queue(~id="", ~queue=[])
  queue = ref(queue)
  def next()
    if not list.is_empty(!queue) then
      r      = list.hd(!queue)
      queue := list.tl(!queue)
      log.info(label=id, "Next song will be #{request.uri(r)}.")
      r
    else
      null()
    end
  end
  def push(r)
    log.info(label=id, "Pushing request #{r} on the queue.")
    queue := list.append(!queue, [r])
  end
  s = request.dynamic(next)
  s.{push=push}
end
```

Internally, it maintains a reference on a list called `queue`. The `next` function pops the first element of the list and returns it, or `null` if the queue is empty, and the `push` function adds a new request at the end of the list. Finally, the source is created by `request.dynamic` with `next` as function returning the next request. Finally, the source is returned, decorated with the method `push`.

Protocols. We have seen that playlists can contain files which are either local or distant, the latter beginning by prefixes such as "`http:`" or "`ftp:`". A *protocol* is a way of turning

such a prefixed URI into an actual file. Most of the time it will consist in downloading the file in the appropriate way, but not only. Liquidsoap supports many protocols and even the possibility of adding your own.

For instance, the `youtube-dl` protocol allows the use of the `youtube-dl` program in order to download files from YouTube.

```
s = single("youtube-dl:https://www.youtube.com/watch?v=TCd6PfxOy0Y")
```

when playing such a file, we need to do more than simply connect to some particular location over the internet, and have to do tricky stuff in order to fetch the video from YouTube. Similarly, the `say` protocol uses the text-to-speech software `text2wav` provided by the festival project in order to synthesize speech. For instance,

```
s = single("say:Hello world!")
```

Incidentally, the `prefix` parameter of `playlist` can be used to add a prefix to every element of the playlist, which is typically useful for protocols. This means that the following will read out the paths of the files in the playlist:

```
s = playlist(prefix="say:","playlist")
```

Another very useful protocol is `annotate` which adds metadata to the following song, as in

```
annotate:artist=The artist,comment=Played on my radio:test.mp3
```

In particular, this can be used to add metadata in playlists (which can contain files beginning with `annotate:`), to specify the usual information such as artist and title (although those are generally already present in files), but also internal information such as cue in and cue out time.

The process protocol. More powerful, the `process` protocol allows to launch any command in order to process files. The syntax is

```
process:<ext>,<cmd>:uri
```

where `<ext>` is the extension of the produced file, `<cmd>` is the command to launch and `uri` is the URI of a file. In the string `<cmd>`, the substring `$(input)` will be replaced by the input file and `$(output)` by the output file (a temporary file whose extension is `<ext>`). For instance, we can convert a file `test.mp3` in stereo WAV (even if the input file is mono) by:

```
s = single("process:wav,ffmpeg -y -i $(input) -ac 2 $(output):test.mp3")
```

When playing it, Liquidsoap will first download `test.mp3` into some place (say `/tmp/temp.mp3`) and then execute

```
ffmpeg -y -i /tmp/temp.mp3 -ac 2 /tmp/temp.wav
```

which will convert it to stereo WAV, and then play the resulting temporary file `/tmp/temp.wav`. The protocol `process` also accepts files of the form

```
process:<ext>,<cmd>
```

in which case only `$(output)` will be replaced in the command. For instance, the implementation of text-to-speech in Liquidsoap essentially amounts to doing

```
s = single("process:wav,echo 'Hello world!' | text2wave > $(output)")
```

which will run

```
echo 'Hello world!' | text2wave > /tmp/temp.wav
```

and play the resulting file.

Registering new protocols. One of the most powerful features of Liquidsoap is that it gives you the ability of registering your own protocols. For instance, suppose that we have a program `find_by_artist` which takes as argument the name of an artist and prints a list of music files from this artist. Typically, this would be achieved by looking into a database of all your music files. For instance, we suppose that executing

```
find_by_artist Halliday
```

will print a list of files such as

```
/data/mp3/allumer_le_feu.mp3
/data/mp3/l_envie.mp3
/data/mp3/que_je_t_aime.mp3
```

We are going to define a new protocol named `artist` so that, when playing a file such as `artist:Halliday`, Liquidsoap will run the above command in order to find a song. This can be done by using the `add_protocol` operator: its first mandatory argument is the name of the protocol (here, `artist`) and the second one is a function which takes as arguments

- a function `rlog` to log the resolution process (you can use it to print whatever is useful for you to debug problems during the generation of the file),
- a duration `maxtime` in seconds which the resolution should not exceed (you should be careful about it when querying distant servers which might take a long time for instance),
- the request,

and returns a list of file names corresponding to the request. Liquidsoap will play the first file available in this list, and those file names might actually also use Liquidsoap protocols. In our example, we can implement the protocol as

```
def artist_protocol(~rlog, ~maxtime, arg) =
  rlog("Finding songs of #{arg}.")
  process.read.lines("find_by_artist #{string.quote(arg)}")
end
add_protocol("artist", artist_protocol,
             doc="Find songs by artist.",
             syntax="artist:<arist name>")
```

We use the `add_protocol` to register our protocol `artist`, where the function `artist_protocol`, which returns the list of files corresponding to a request, simply returns the list of all the files printed by the command `find_by_artist`. The `doc` parameter is free form documentation for the protocol and the `syntax` parameter provides an illustration of a typical request using this protocol (both are only for documentation purposes). Once this defined, we can finally play songs of any artist by performing requests of the form

```
s = single("artist:Sinatra")
```

and, of course, an URI such as "`artist:Nina Simone`" could also be used in a playlist or pushed in a request queue.

Soundcard inputs. In order to input sound from a soundcard, you should use functions such as `input.alsa` or `input.pulseaudio` or `input.portaudio` depending on the library you want to use for this: the first one is a little more efficient, because it is closer to the hardware, and the second is more portable and widespread. For instance, you can hear your voice with

```
s = buffer(input.alsa())
output(s)
```

Basically, this script plays the stream generated by `input.alsa`. However, we have to use the `buffer` operator in order to bufferize the stream coming from the soundcard and deal with synchronization issues between the input and the output, as detailed in there.

If you want to use a particular device, you should use the parameter `device` of `input.alsa`, which takes as argument a string of the form `hw:X,Y` where `X` is the card number and `Y` is the device number. The list of all devices available on your computer can be obtained with the command

```
aplay -l
```

On my laptop this returns

```
card 0: PCH [HDA Intel PCH], device 0: ALC3246 Analog [ALC3246 Analog]
  Subdevices: 0/1
  Subdevice #0: subdevice #0
card 0: PCH [HDA Intel PCH], device 3: HDMI 0 [HDMI 0]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

so that if I want to input from HDMI (the second one listed above), I should use `hw:0,3` as `device` parameter.

By default, the ALSA operators have an internal buffer in order to be able to cope with small delays induced by the soundcard and the computer. However, you can set the `bufferize` parameter to `false` in order to avoid that in order to reduce latencies. For instance, if you are lucky, you can hear your voice almost in realtime, with some flanger effect added just for fun:

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

Beware that by reducing the buffers, you are likely to hear audio glitches due to the fact that blank is inserted when audio data is not ready soon enough. In this case, you should also see the following in the logs

```
Underrun! You may minimize them by increasing the buffer size.
```

which indicates that you should buffer in order to avoid defects in the audio.

Distant inputs with harbor. Many programs are able to stream to an Icecast server, and we can use those as an input for Liquidsoap scripts with the `input.harbor` operator. This operator instructs Liquidsoap to run an Icecast-compatible server, called *harbor*. Clients can then connect to it, as they would do on any Icecast server, and the stream they send there is then available as a source in your script. This can be useful to relay an intermittent live stream without having to regularly poll the Icecast server to find out when it is available. It can also be used to have interventions from distant live speakers and DJs: for instance the Mixxx software can be used to easily make mixes from home. A typical setup would be

```
playlist = playlist("~/Music")
live     = input.harbor("live", port=8000, password="hackme")
radio    = fallback(track_sensitive=false, [live, playlist])
output(radio)
```

In this example, we use the `playlist` source by default, but we give priority to the `live` source, which is a harbor input, available only when some client connects to it. Apart from the parameters specifying the port and the password to use when connecting to the server, the unlabeled argument specifies the *mountpoint*: this should be specified by the client when connecting, which means that a same harbor server can simultaneously relay multiple sources, as long as they use different mountpoints.

Sending sound to harbor. In order to test the above script, we need some software which can send streams using the Icecast protocol to the harbor input. Ours obviously being Liquidsoap, after having started the above script, you can run the second script

```
s = mksafe(playlist("~/Music"))
output.icecast(host="localhost", password="hackme", mount="live",
               %mp3, s)
```

which will stream connect to the harbor Icecast server and stream our music library in MP3 format. Of course, `localhost` should be changed by the name (or IP) of the server if you are not running the client on the same machine.

Another possibility would be to use the `shout` program which can be used to directly send music files to Icecast. For instance,

```
cat test.ogg | shout --format ogg --host localhost --pass hackme --mount /live
```

will send the file `test.ogg` to our harbor server.

Yet another possibility consists in using the `darkice` program which captures the microphone of the computer and sends it to an Icecast server. We can use it to stream our voice with

```
darkice -c darkice.cfg
```

where `darkice.cfg` is a configuration file specifying the parameters of the program, such as

```
[general]
duration        = 0
bufferSecs      = 5

[input]
device          = pulseaudio
sampleRate      = 44100
bitsPerSample   = 16
channel         = 2
paSourceName    = default

[icecast2-0]
format          = mp3
bitrateMode     = abr
bitrate         = 128
quality         = 0.8
server          = localhost
port            = 8000
password        = hackme
mountPoint      = live
```

Securing harbor. Since harbor exposes a server to the internet, you should be serious about security and think thoroughly about who should be having access to this server. The fact that the server is programmed in OCaml makes it quite unlikely that an exploit such as a buffer overflow is possible, but one never knows.

First, the list of IP which are allowed to connect to harbor can be changed with the following setting:

```
set("harbor.bind_addrs", ["0.0.0.0"])
```

It takes as argument a list of allowed IP, the default one `0.0.0.0` meaning that every IP is allowed.

In practice, it is often quite difficult to know in advance the IP of all the clients, so that the main security is given by the password which is passed as argument of `input.harbor`: please choose it wisely, and avoid at any means the default password "`hackme`". Even with a strong password, the security is not very good: if some client leaks the password or you want to revoke a client, you have to change it for every client which is not convenient. For this reason, the authentication can also be done through a function, which is passed as the `auth` argument of `input.harbor` and is of type

```
({user : string, password : string, address : string}) -> bool
```

It takes as argument a record containing the username, the password and the IP address of a client trying to log in and returns whether it should be allowed or not. Typically, you would like to call an external script, say `harbor-auth`, which will take the username and password as argument and print "`allowed`" if the user is allowed (such a command would usually look into a database to see whether the credentials match, and perhaps do additional checks such as ensuring that the user has the right to connect at the given time).

```
def auth(login)
  ans = process.read.lines("./harbor-auth \
        #{string.quote(login.user)} #{string.quote(login.password)}")
  if list.hd(default="", ans) == "allowed" then
    true
  else
    log.important("Invalid login from #{login.user}")
    false
  end
end
s = input.harbor("live", port=8000, auth=auth)
output(s)
```

Here, our function `auth` begins by executing the script `harbor-auth` with the username and password as argument. Note that we use `string.quote` to escape shell special characters, so that the user cannot introduce shell commands in his username for instance… The `process.read.lines` function returns the list of lines returned by our script and our function returns `true` or `false` depending on whether this first line is "`allowed`" or not. In this way you could easily query an external database of allowed users.

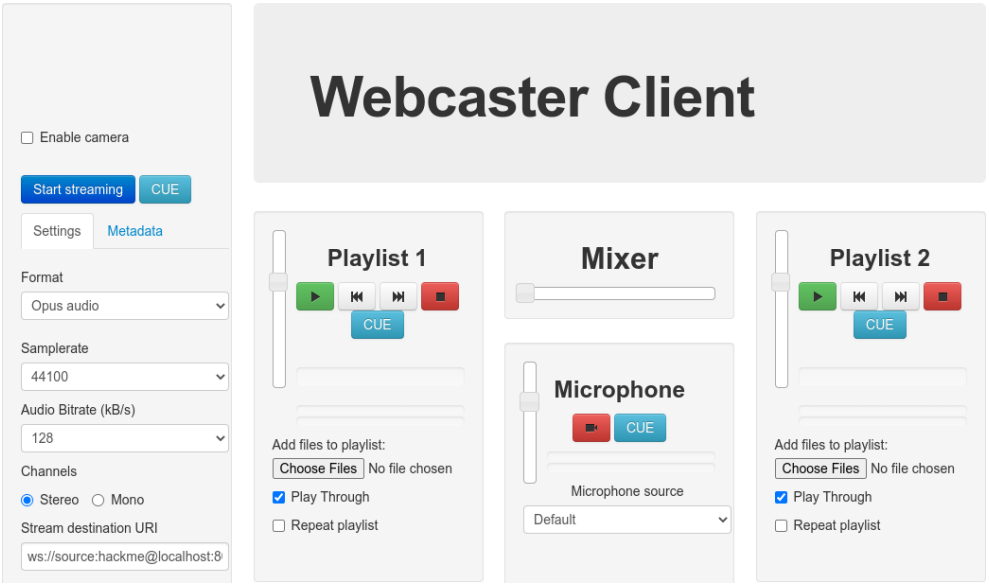
Finally, the clients should be able to determine that they are really connected with your server and not some hacker's one. The best way to achieve that is to use SSL certificates, which can be handled with the `input.harbor.ssl` variant of the harbor source, which is present only if Liquidsoap has been compiled with SSL support. The certificate can be specified with the setting `harbor.ssl.certificate` (the setting `harbor.ssl.private_key` can also be used to specify the private key and `harbor.ssl.password` the 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, which you can for instance obtain from *Let's Encrypt*. Alternatively, 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
```

Webcast. For interventions by external collaborators, it can be problematic to have them perform a "complex setup" such as installing Liquidsoap or run `darkice`. For this reason, we have developed a protocol which allows streaming sound directly from the browser, be it from music files or the microphone, without having to install anything: this protocol is called *webcast* and we provide a JavaScript implementation `webcast.js`, which should be easy to integrate in your own website, based on websockets and media recorder API. Of course, `input.harbor` has support for it and will accept clients using the webcast protocol in addition to the traditional Icecast protocol.

A full-fledged website based on this library, called *Webcaster*, is available. It provides a web interface which looks like this:



As you can see, it allows performing simple DJ interventions, by mixing between two playlists and the microphone input. And should you need more, the source code is available. In the lower-left corner is the URI we should connect to, in the following form:

```
ws://username:password@host:port/mountpoint
```

For instance, with the above harbor input example, we should thus use the URI

```
ws://user:hackme@localhost:8000/live
```

(the username is not taken in account when using the basic password authentication).

External inputs. In case you are still not satisfied (for instance, if you have very specific needs or are the proud owner of hardware which is not widely supported), it is possible to use any program as an input, as long as this program echoes audio on its standard output either in WAV or in raw audio format, with the operators `input.external.wav` and

`input.external.rawaudio`. Here, "raw audio" data consists in interleaved samples encoded as signed 16 bits little-endian integers (the use of the WAV format is preferable to avoid confusions about the encoding format). For instance, we can play a sine wave which is generated by the program `ffmpeg` with

```
cmd = "ffmpeg -f lavfi -i sine=frequency=440 -ac 2 -f wav -"
s   = input.external.wav(cmd)
```

or with

```
cmd = "ffmpeg -f lavfi -i sine=frequency=440 -ac 2 -f s16le -"
s   = input.external.rawaudio(cmd)
```

The argument `cmd` is the program which is going to be executed: here, we use `ffmpeg`, which is instructed to generate a sine at 440 Hz (`-i sine=frequency=440`) in stereo (`-ac 2`), encode it in WAV (`-f wav`) or raw format (`-f s16le`), and output it on the standard output (`-`).

FFmpeg input. The above examples are only to illustrate the use external programs as input, but you would not use it with `ffmpeg` in practice because Liquidsoap has builtin support for it, through the `input.ffmpeg` operator. A more natural way of writing the above to generate a sine through FFmpeg would thus be

```
s = input.ffmpeg(format="lavfi", "sine=frequency=440")
```

GStreamer input. Finally, another very general possibility for input is to use the `input.gstreamer.audio` operator in order to use the GStreamer library to generate audio. The generation itself is described through a *pipeline* which consists in a sequence of GStreamer operators separated by "`!`": a pipeline "`a ! b`" means that the output of operator "`a`" should be fed to operator "`b`". We refer the reader to the documentation of the library for more information about it. In Liquidsoap, the pipeline can be passed in the argument labeled `pipeline`, as expected. For instance, we can generate a sine wave (again) with

```
s = input.gstreamer.audio(pipeline=
      "audiotestsrc ! audioamplify amplification=1.5")
```

where we use the operator `audiotestsrc` to generate a sine, which we pipe to the `audioamplify` operator to change its volume. Similarly, we can play the file `test.mp3` with

```
s = input.gstreamer.audio(pipeline='filesrc location="test.mp3"')
```

In practice, no one would use the above example as is, because Liquidsoap already has builtin support for using GStreamer to decode files…

JACK input. If the other program has support for it, it is possible to use JACK with the `input.jack` operator. This library is dedicated to the communication of audio data between programs and greater stability and precision is expected than with the above method. Its use is detailed in there.

SRT. In order to transmit a stream between two machines on the same local network, one can use the Icecast protocol (i.e. `input.harbor` or `input.http`) but this is not satisfactory: firstly, the has to be compressed (the Icecast protocol does not allow sending uncompressed data such as WAV, it can however send lossless compressed data with the FLAC codec) and, secondly, it induces much delay in the stream due to the various buffers used by Icecast.

A much better solution consists in using the SRT (for *Secure Reliable Transport*) protocol, which allows reliable data transmission with low latency, and can be performed using the `input.srt` operator. A minimal script could be

```
s = input.srt()
output.pulseaudio(fallible=true, buffer(s))
```

Note that we need to use the `buffer` operator here because SRT uses its own synchronization mechanism, which is different from the one on the output based on the Pulseaudio library, see below. The input is not available unless it receives some stream, which is why we pass `fallible=true` to `buffer`. We can send such a stream with `ffmpeg` for instance: the script

```
ffmpeg -re -i test.mp3 -f mp3 -c:a copy srt://localhost:8000
```

sends the contents of the `test.mp3` using SRT on the host `localhost` (you should of course change this if you are connecting to a distant machine) on the port 8000, which is the default one. The `-re` option ensure that the file is sent progressively and not all at once. Another option to send an SRT stream is, of course, to use Liquidsoap. We can use the script

```
s = playlist("~/Music")
output.srt(fallible=true, host="localhost", %wav, s)
```

to stream our music library to the above SRT input in WAV format (you could use `%mp3` instead of `%wav` to compress in MP3 in order save bandwidth).

6.2 Scheduling

Now that we have a wide panel of sources, we need to combine them.

Fallback. The first way of combining sources is through the `fallback` operator, which takes as argument a list of sources, and plays the first one which is available, i.e. can produce some stream. We have already seen examples of this with request queues (here) such as

```
radio = fallback([queue, playlist])
```

Here, we want to play a song from the request queue when there is one, otherwise we play songs from the playlist. By default, if we are playing a song from the playlist and there is a new song in the queue, the operator will wait for the current playlist song to finish before playing the one from the queue. This behavior can be changed by setting the `track_sensitive` parameter to `false`, in which case the song from the queue will be immediately played:

```
radio = fallback(track_sensitive=false, [queue, playlist])
```

Typically, you would use this to switch to a live show when available

```
playlist = playlist("~/Music")
live     = input.harbor("live", port=8000, password="hackme")
radio    = fallback(track_sensitive=false, [live, playlist])
```

or to feature live interventions when someone is speaking on the microphone

```
mic   = buffer(input.alsa())
mic   = blank.strip(max_blank=2., min_noise=.1, threshold=-20., mic)
music = playlist("~/Music")
radio = fallback(track_sensitive=false, [mic, music])
```

In this last example, we are using the operator `blank.strip` to make the source `mic` unavailable when there are at least 2 seconds of silence (the duration is controlled by the `max_blank` argument, and the `threshold` argument indicates that we consider anything below -20 dB as silence): in this case, the `fallback` operator will default to the `music` playlist until someone is speaking again.

In order to make a source `s` always available, it is quite common to stream blank when the source is not available, i.e. re-define the source

```
s = fallback(track_sensitive=false, [s, blank()])
```

with a fallback on blank. Since this is quite common in scripts, the function `mksafe` is defined in the standard library as a shorthand, and the above is equivalent to writing

```
s = mksafe(s)
```

Skipping fallback. In a fallback situation where we have a live source and a music playlist as above, the solution we provided is not entirely satisfactory: when the live source ends, the music source is played again where we left it, whereas it is often the case that we want to start on a fresh track. The `fallback.skip` operator allows precisely this: the script

```
radio = fallback.skip(live, fallback=music)
```

will play the `live` source and default to the `music` source, like a regular track insensitive fallback operator, but it will also skip the current track of the `music` source after switching to the `live` source, so that we will begin on a fresh track when switching back again to `music`.

For didactic purposes, let us provide another way of implementing this. The `on_leave` method of a source allows registering a function which is to be called when the source is not used anymore. We can use this to enforce skipping the `music` source when we switch the `live` source, and the following script will behave like the above one:

```
music.on_leave(music.skip)
radio = fallback(track_sensitive=false, [live, music])
```

Switching and time predicates. Another way of selecting a source among multiple ones is the `switch` operator. It takes as argument a list, whose elements are pairs consisting of a predicate and a source. Here, each *predicate* is function taking no argument an returning a boolean (it is of type `() -> bool`) indicating whether the corresponding source should be played or not: a typical predicate is `{8h-10h}` which is true when the current time is between 8h and 10h. The `switch` operator will select and play the first source whose predicate returns `true`.

For instance, supposing that we have two different playlists for night and day, we could alternate between those depending on the hour with

```
radio = switch( [ ({0h-7h}, night), ({7h-24h}, day) ] )
```

Here also, the `track_sensitive` parameter controls whether a change of source only occurs at track boundaries (when `true`, which is the default) or as soon as possible.

If you want to make sure that there is always something to play, the condition of the last source should always be true, i.e. you can use `{true}`. For instance,

```
radio = switch([
    ({6h-9h},   morning),
    ({20h-24h}, evening),
```

```
    ({true},    music)
  ])
```

will have two special programs on the morning and the evening, and will default to the `music` playlist at other times. We thus obtain the same behavior as if we had used a `fallback` operator:

```
radio = switch([({6h-9h}, morning), ({20h-24h}, evening)])
radio = fallback([radio, music])
```

Time predicates. In the above examples, `{0h-7h}` is a *time predicate*: it is something which is `true` or `false` depending on the current time. Some other examples of time predicates are

`{11h15-13h}`	between 11h15 and 13h
`{12h}`	between 12h00 and 12h59
`{12h00}`	at 12h00
`{00m}`	on the first minute of every hour
`{00m-09m}`	on the first 10 minutes of every hour
`{2w}`	on Tuesday
`{6w-7w}`	on weekends

Above, `w` stands for weekday: 1 is Monday, 2 is Tuesday, and so on. Sunday is both 0 and 7.

Other predicates. We could also use this operator to manually switch between sources. As an illustration, supposing that we have two radio streams named `radio1` and `radio2`, we could use a script such as

```
radio = switch(track_sensitive=false, [(p, radio1), ({true}, radio2)])
```

where the predicate `p` determines when `radio1` should be played. For instance, if we want to play it when a file `select-radio` contains "`1`", we could define it as

```
p = {file.contents("select-radio") == "1"}
```

Another way to achieve this could be use an "interactive boolean", as detailed in there, and defined instead

```
set("server.telnet", true)
p = interactive.bool("r1", false)
```

The interactive boolean is a sort of reference whose value can be changed over the telnet by issuing commands such as "`var.set r1 = true`", which sets the value of the boolean named `r1` to `true`. Therefore, we can switch to radio 1 by typing the command

```
echo "var.set r1 = true" | telnet localhost 1234
```

and back to radio 2 with

```
echo "var.set r1 = false" | telnet localhost 1234
```

(or directly connecting to the telnet server and issuing the commands, see there).

Adding. Instead of switching between two sources, we can play them at the same time with the `add` operator, which takes a list of sources whose sound are to be added. For instance, if we want to make a radio consisting of a microphone input together with background music (which is often called a "bed"), we can define

```
radio = add([mic, bed])
```

This will play the two sources `mic` and `bed` at equal volume. By default, the volume of the output is divided by 2 (because there are 2 sources) in order not increase the loudness too much. If you want to keep the original volume of the sources, you should set the `normalize` parameter to `false`:

```
radio = add(normalize=false, [mic, bed])
```

but beware that this might result into some clipping if the two sources are loud, which is never nice to hear, see also there.

As a side note, the operator `add` only adds the sources which are ready. This means that if `mic` is taken from a harbor input such as

```
mic = input.harbor("mic")
```

and the client did not connect or was disconnected, we will hear only the bed, as expected.

Weights. The `add` operator also offers the possibility of weighting the sources, i.e. specifying their relative volume: if we want to hear the microphone twice as loud as the bed, we should give the microphone twice the weight of the bed. The weight of each source can be specified in a list passed in the `weights` arguments. For instance,

```
radio = add(weights=[2., 1.], [mic, bed])
```

assigns the weight `2.` to `mic` and `1.` to `bed`. This is equivalent to amplifying each of the sources with the corresponding factor, i.e.

```
radio = add([amplify(2., mic), amplify(1., bed)])
```

but more efficient and natural.

Sequencing. The `sequence` operator allows a behavior which sometimes useful: it takes as argument a list of sources and plays one track from each source in order, and finally keeps on playing tracks from the last source. This means that

```
s = sequence([s1, s2, s3])
```

will play one track from `s1`, one track from `s2` and will then keep on playing `s3`. We will for instance use this in section below in order to insert jingles during transitions.

Jingles and ads. Jingles are short announcements, generally indicating the name of the radio or the current show. They are quite important in order for the listener to remember the brand of your radio and create some familiarity with the radio (the music changes, but the jingles remain the same). Technically, jingles are not different from any other music source, but we give here the usual ways of inserting those, presenting tricks which might be useful in other situations too (in particular, ads follow basically the same techniques). We suppose here that we have a source `music` which plays our music and a source `jingles` which plays jingles: typically, it will be defined as

```
jingles = playlist("jingles")
```

where `jingles` is a playlist containing all our jingles.

Rotating tracks. The most basic strategy consists in inserting one jingle every n tracks, which is easily achieved thanks to the `rotate` operator. It takes a list of sources and a list of weights associated to each source (in the argument labeled `weight`), and selects tracks from the sources according to the weights. For instance, in the following script

```
radio = rotate(weights=[1, 4], [jingles, music])
```

we play `jingles` with weight `1` and `music` with weight `4`: this means that we are going to play one jingle, then four music tracks, then one jingle, then four music tracks, and so on.

If you want something less regular, the `random` operator can be used instead of `rotate`:

```
radio = rotate(weights=[1, 4], [jingles, music])
```

It is basically a randomized version of the previous source, which will randomly chose tracks from `jingles` and `music`, the probability of choosing a track from the latter being four times the probability of choosing a track from the former.

The `rotate` and `random` operators can also be used to vary the contents of a source. For instance, if we wanted our `jingles` sources to play alternatively a jingle, a commercial and an announcement for a show, we could have defined

```
jingles = rotate([
    playlist("jingles"),
    playlist("commercials"),
    playlist("announcements")
  ])
```

Playing jingles regularly. Another approach for jingles consists in playing them at regular time intervals. This is easily achieved with the `delay` operator function which prevents a source from being available before some time. For instance, we can play a jingle roughly every 30 minutes with

```
radio = fallback([delay(1800., jingles), music])
```

Above, the function `delay` above enforces that, after playing a track, the `jingles` source will not be available again before 1800 seconds, which is 30 minutes. Therefore, every time the current music track ends and more than 30 minutes has passed since the last jingle, a new one will be inserted. As a variant, we can add the jingle on top of the currently playing music with

```
radio = add([delay(1800., jingles), music])
```

Jingles at fixed time. Instead of inserting jingles regularly, you might want to insert them at fixed time. This is quite a common approach, but a bit tricky to achieve. Suppose that we want to play a jingle at the beginning of each hour, without interrupting the current track. One would typically write a script such as

```
radio = switch([
    ({00m},  jingles),
    ({true}, music)
  ])
```

which states that when the current minute of the time is "00", we should play the `jingles` source. But this is not really good: if a track from the music source starts at 11h58 and ends at 12h01 then no ad will be played around noon. In order to accommodate for this, we are tempted to widen the time predicate and replace the second line with

```
    ({00m-15m}, jingles),
```

Well, this is not good either: if a track of the music source ends at 12h01, we now hear a jingle as expected, but we actually continuously hear jingles for 14 minutes instead of hearing only one. In order to fix this, we are tempted to use the `once` operator and change the line to

```
    ({00m-15m}, once(jingles)),
```

This is not good either: `once(jingles)` plays only one track from `jingles`, but during the whole execution of the script. This means that our script will only work as expected on the first hour, where we will correctly hear one jingle, but on the following hours we will hear no jingle because one has already been played. An acceptable solution consists in using `delay` to ensure that we should wait at least 30 minutes before playing another jingle and replace the second line by

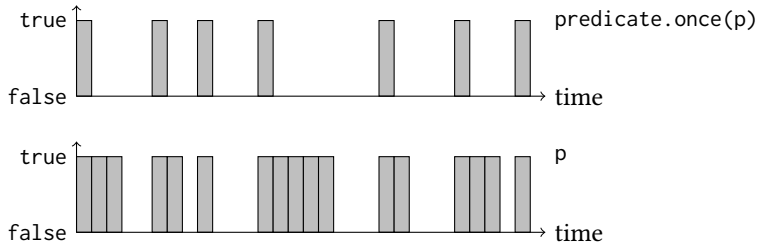
```
    ({00m-15m}, delay(1800., jingles)),
```

it says that we should play a jingle in the first quarter of every hour and not more often than once every half hour, and achieves precisely what we want. Note that anything from `900.` (15 minutes) to `2700.` (45 minutes) would be acceptable as argument for the `delay`.

As a last remark, if we want to play the jingle exactly at the top of the hour, and interrupt the currently playing song if necessary, it is enough to add `track_sensitive=false` to the `switch` operator:

```
radio = switch([
    (predicate.once({00m-15m}), jingles),
    ({true},                    music)
  ])
```

Jingles at fixed time: alternative approach. In the previous example, an alternative approach instead of using the `delay` operator, consists in using the `predicate.once` function. It takes a predicate `p` as argument, and returns a predicate which is true only once each time `p` is continuously true. In case it helps, we have illustrated in the following figure an example of a predicate `p` over time (below) and the resulting predicate `predicate.once(p)` over time (above):



This means that `predicate.once({00m-15m})` is a predicate which is true once between in the first quarter of every hour, and can thus be used to play on jingle in the first quarter of every hour as follows:

```
radio = switch([
    (predicate.once({00m-15m}), jingles),
    ({true},                    music)
  ])
```

As a variant, if we wanted to play a jingle every half hour, we could replace the second line by

```
    (predicate.once({00m-15m or 30m-45m}), jingles),
```

As another variant, if we wanted to play 3 jingles, we could write

```
    (predicate.at_most(3, {00m-15m}), jingles),
```

where `predicate.at_most` is similar to `predicate.once`, but is true a given number of times instead of only once (its it pointless to play 3 jingles in a row, but this can be quite useful for ads for instance).

If we want to add the jingle on top of the currently playing music, we can use the function `source.available` which takes as arguments a source and a predicate, and makes the source available only when the predicate is satisfied. We can then add the music with the jingles source made available once every half hour as follows:

```
radio = add([
    source.available(jingles, predicate.once({0m-10m or 30m-40m})),
    music
  ])
```

Signaling. As a more advanced use of predicates, we would like to introduce the `predicate.signal` function which creates a particular predicate which is false most of the time, unless some other part of the program sends a "signal", in which case the predicate becomes true once and then false again, until the next signal. Concretely, we can use this function to create a predicate `p` by

```
p = predicate.signal()
```

The predicate `p` is false by default, in order to make it true we can send a signal by calling its method `signal`, i.e. writing `p.signal()`. For instance, in the script

```
p = predicate.signal()
radio = switch([(p, jingles), ({true}, music)])
thread.run(every=1200., {p.signal()})
```

we use `predicate.signal` to create a predicate `p` which controls playing jingles (if there is no jingle to play we default to the `music` source). We then use `thread.run` to execute `p.signal()` every 20 minutes (or 1200 seconds). This means that every 20 minutes (or a bit more because we are waiting for the current track of `music` to end), we will hear a jingle.

Of course, this is not the way one would typically regularly insert jingles, but this can easily be modified to insert jingles by interacting with the environment. For instance, we can register a command on the telnet server as follows:

```
def cmd(_)
  p.signal()
  "Jingle inserted!"
end
server.register("insert_jingle", cmd)
```

The function `cmd`, when called with some argument, will run `p.signal()` and return a string indicating that the jingle has been inserted. We then use the `server.register` function to instruct Liquidsoap that the function `cmd` should be called when a user runs the command `insert_jingle` on the telnet server. This means that if you connect to the telnet server and

type `insert_jingle`, a jingle will be inserted after next track, which could be quite useful if you are designing some sort of graphical interface for your radio.

Inserting jingles on metadata. We will also see in next section that the insertion of jingles can also conveniently be triggered by metadata in sources.

Inserting jingles in transitions. Yet another method for inserting transition consists in adding them in transitions between tracks, this is detailed in a later section.

6.3 Tracks and metadata

Liquidsoap has a notion of *track* in stream, which is generally used to mark the boundary between two successive songs. We have seen that many functions to control the stream (`fallback`, `switch`, etc.) have the ability to detect tracks and only change stream when a track is over in order not to abruptly interrupt a playing song (this behavior can be altered by setting the `track_sensitive` parameter to `false`).

To every track is associated metadata, which is information concerning the song which is going to be played. In Liquidsoap, metadata can actually be present at any time, and does not have to correspond to a track boundary (we can have metadata in the middle of a song) although this is generally the case. We have seen in there that metadata is generally coded as an association list: this is a list consisting of pairs of strings, whose type is thus `[string * string]`. Typically, the metadata `m` for a track will look like

```
m = [("artist", "Sinatra"), ("title", "Fly me")]
```

which indicates that the artist is "Sinatra" and the title is "Fly me". Typical metadata fields are: `artist`, `title`, `album`, `genre`, `year` and `comment`.

Manipulating metadata. In order to retrieve the title in such a list, one can use the notation

```
m["title"]
```

which returns the value associated to the field `title` in the metadata `m`, the empty string `""` being returned in the case where the metadata is not present. Changing the value of some metadata is simply obtained by putting the new metadata at the top, by using the function `list.add` or `list.append`. For instance, we can define a metadata `m'` where the artist has been changed and the year has been added by

```
m' = list.append([("artist", "S"), ("year", "1964")], m)
```

or, if we only want to change the year,

```
m' = list.add(("year", "1964"), m)
```

Metadata in requests. Metadata are usually stored within files: for instance, MP3 files generally contain metadata encoded in the ID3v2 format. Typical operators reading files, such as `playlist`, automatically read those when opening a file. We recall that it is also possible to add metadata to files in playlists using the `annotate` protocol. For instance,

```
annotate:artist=The artist,comment=Played on my radio:test.mp3
```

Incidentally, the `prefix` parameter of the `playlist` operator can be used to add a prefix to every file in the playlist. It can in particular be used to annotate every file in order to add

some metadata. For instance, if we want to set the metadata `jingle` to `true` for every track in our playlist, we can write something like

```
s = playlist(prefix="annotate:jingle=true:", "~/Music")
```

Handling tracks. Every source has `on_track` and `on_metadata` methods, which respectively enforce the execution of a function when a track boundary or metadata occur in the stream. In both cases, the method takes as argument a function of type

```
([string * string]) -> unit
```

This function will itself be called with the metadata as argument when a track or metadata occurs.

Logging tracks. We can for instance use this mechanism to log every song which has gone on air:

```
def log_song(m)
  artist = m["artist"]
  title  = m["title"]
  file.write(append=true, data="#{artist} - #{title}\n", "/tmp/songs")
end

radio = playlist("~/Music")
radio.on_track(log_song)
```

We first define a function `log_song` which takes the metadata `m` as argument, extracts the artist and the title, and appends those to the file `/tmp/songs`. We then run the method `on_track` of our `music` source to register this function to be called when there is a new track and that's it! By the way, if you want a quick and effective way of logging the metadata, we advise the use the `json.stringify` function, which will convert all the metadata at once into a standardized textual representation in JSON format:

```
def log_song(m)
  file.write(append=true, data="#{json.stringify(m)}\n", "/tmp/songs")
end
```

Logging the next track. It is sometimes convenient to store the metadata for the next song to be played, for instance to announce it on a website. This is difficult in general because Liquidsoap does not compute much of the stream in advance. However, if you are using a playlist, this can be achieved as follows:

```
def log_next(r)
  m = request.metadata(r)
  file.write(data="#{json.stringify(m)}", "/tmp/next-song")
  true
end

radio = playlist("~/Music", length=40., check_next=log_next)
```

Here, we are (ab)using the `check_next` argument which specifies a function which is called in order to validate the next song: we register our `log_next` function which always validates the next song (it always returns `true`), but logs the metadata of the song in between. By default, the duration of a song is not computed precisely and estimated to 30 seconds (this can be changed with the `default_duration` parameter of `playlist`). We use the `length` parameter

to request that we always have at least 40 seconds of estimated remaining duration (instead of 10 by default), which forces the `playlist` operator to always prepare one song in advance, which will be validated by our `log_next` function, thus providing us with its metadata.

Adding jingles on metadata. The functions `on_track` and `on_metadata` can also be used in order to insert jingles (or ads) when some metadata is present. For instance, we suppose that we have a music source `s`, perhaps generated by a playlist: when a track has the metadata `jingle` set to `true`, we want to play a jingle beforehand. One way to perform this is

```
q = request.queue()
def insert_jingle(m)
  log.info("Got metadata")
  if m["jingle"] == "true" then
    log.info("Inserting jingle")
    q.push.uri("jingle.mp3")
  end
end
s.on_track(insert_jingle)
s = fallback(track_sensitive=false, [q, s])
output(s)
```

It consists in creating a queue `q` and executing a function `insert_jingle` when a track is present: this function will look whether the value of the `jingle` metadata is `true`, and if this is the case it will insert a jingle (the file `jingle.mp3`) into the queue, which will then be played by a usual fallback mechanism. As an easy variant of the above script, we can read out the last song which was played on air, by inserting in a queue a request to read it using the `say` protocol:

```
q = request.queue()
def insert_title(m)
  q.push.uri("say:Last song was #{m['title']} by #{m['artist']}")
end
s.on_track(insert_title)
s = fallback([q, s])
output(s)
```

Another approach to insert jingles when particular metadata is present could consist in using the `predicate.signal` function detailed above to trigger playing one track of a `jingles` playlist when the metadata is present:

```
jingles = playlist("jingles")
p = predicate.signal()
def insert_jingle(m)
  log.info("Got metadata")
  if m["jingle"] == "true" then
    log.info("Inserting jingle")
    p.signal()
  end
end
s.on_track(insert_jingle)
s = switch([(p,jingles), ({true},s)])
output(s)
```

Prepending and appending tracks. The logic of the above scripts can be somewhat simplified by the use of the `prepend` operator: this operator takes as argument a function which is called on every track, with its metadata as source, and returns a source to be played before the current track. For instance, we can insert a jingle when the metadata `jingle` is set to `true` by:

```
def insert_jingle(m)
  log.info("Got metadata")
  if m["jingle"] == "true" then
    log.info("Inserting jingle")
    jingles
  else
    source.fail()
  end
end
s = prepend(s, insert_jingle)
output(s)
```

The function `insert_jingle` looks at the metadata, and if present returns the `jingles` source, containing all the jingles, of which one track will be played. If the metadata is not present, we return `fail()` which is a source which is never available: in this case, prepend will simply not insert any track because none is ready. The function is then registered with the `prepend` operator.

Of course, there is a dual operator `append` which allows appending a track for every track: contrarily to `prepend`, it inserts the track after the currently playing track. For instance, we can read the song which we have just played with

```
def insert_title(m)
  single("say:Last song was #{m['title']} by #{m['artist']}")
end
s = append(s, insert_title)
output(s)
```

Rewriting metadata. If you want to systematically modify the metadata, you can use the `map_metadata` function which will modify the metadata of a source: it takes as argument a function and a source, and uses the function to systematically change the metadata. The type of the function is

```
([string * string]) -> [string * string]
```

it takes the current metadata and returns the new metadata to be inserted. For instance, we can add the year in the title and a comment into the metadata of our source `s` with

```
def update_metadata(m)
  [
    ("title",   "#{m['title']} (#{m['year']})"),
    ("comment", "Encoded by Liquidsoap!")
  ]
end
s = map_metadata(update_metadata, s)
```

Whenever a metadata passes on the source `s`, the function `update_metadata` is executed with it and returns the metadata to insert. Here, it states that we should set the title to "<title> (<year>)" (where <title> is the title and <year> is the year present in the original metadata) and that we should advertise about Liquidsoap in the `comment` field.

As another example, suppose that your files do not have proper metadata, for instance, there could be no metadata at all. Even in this case, Liquidsoap will add some metadata in order to indicate internal information such as the filename, more details are given in there. We could thus use the filename as the title as follows:

```
def update_metadata(m) =
  title = path.remove_extension(path.basename(m["filename"]))
  [("title", title)]
end
s = map_metadata(update_metadata, s)
```

The function `path.basename` gives the filename without the leading path and `path.remove_extension` removes the extension of the file.

Removing tracks and metadata. In order to remove the tracks indications from a source, the `merge_track` operator can be used: it takes a source `s` as argument and returns the same source with the track boundaries removed.

Similarly, `drop_metadata` removes all metadata from a source. This can be useful if you want to "clean up" all the metadata before inserting your own, as indicated below.

Inserting tracks and metadata. If you want to insert tracks or metadata at any point in a source, you can use `insert_metadata`: this operator takes a source as argument and returns the same source with a new method `insert_metadata` whose type is

```
(?new_track : bool, [string * string]) -> unit
```

It takes an argument labeled `new_track` to indicate if some track should be inserted along with the metadata (by default, it is not the case) and the metadata itself, and inserts the metadata into the stream. For instance, suppose that we have a source `s` and we want to set the title and artist metadata to "Liquidsoap" every minute. This can be achieved by

```
s = insert_metadata(s)
thread.run(every=60., {
  s.insert_metadata([("artist", "Liquidsoap"),("title", "Liquidsoap")])
  })
```

Here, we add the ability to insert metadata in the source `s` with the operator `insert_metadata`, and we then use `thread.run` to regularly call a function which will insert metadata by calling `s.insert_metadata`.

Similarly, we can add a telnet command to change the title metadata of a source `s` by

```
s = insert_metadata(s)
def cmd(t)
  s.insert_metadata([("title", t)])
  "Title set to #{t}."
end
server.register("set_title", cmd)
```

We begin by defining a function `cmd` which takes the title as argument, inserts it into the stream of `s`, and returns a message saying that the title was inserted. We then register this command as `set_metadata` on the telnet server, as detailed in there: when we enter the command

```
set_title New title
```

on the telnet, the title will be set to "New title". In fact, the standard library offers a generic function in order to do this and we do not have to program this by ourselves: the function `server.insert_metadata` takes an identifier `id` and a source as argument and registers a command `id.insert` on the telnet which can be used to insert any metadata. A typical script will contain

```
s = server.insert_metadata(id="src", s)
```

and we can then set the title and the artist by running the telnet command

```
src.insert title="New title",artist="Liquidsoap"
```

(the argument of the command is of the form `key1=val1,key2=val2,key3=val3,...` and allows specifying the key / value pairs for the metadata).

Skipping tracks. Every source has a method `skip` whose purpose is to skip the current track and go to the next one. For instance, if our main source is `s`, we can hear the first 5 seconds of each track of `s` with

```
thread.run(every=5., {s.skip()})
output(s)
```

We could also easily use this to register a telnet command, but this done by default with the `playlist` operator: you can always use the `skip` telnet command (or `id.skip` if an identifier `id` was specified for the source) to skip the current song. For instance, with the script

```
set("server.telnet", true)
s = playlist(id="music", "~/Music")
output(s)
```

running the telnet command `music.skip` will end the current track and go to the next one. As another example, let us register skip as an HTTP service: the script

```
def skipper(~protocol, ~headers, ~data, uri)
  s.skip()
  http.response(data="The current song was skipped!")
end
harbor.http.register(port=8000, "skip", skipper)
output(s)
```

makes it so that whenever we connect to the URL `http://localhost:8000/skip` the function `skipper` is called and the current song on the source `s` is abruptly ended: this is part of the interaction you would typically have when designing a web interface for your radio. Interaction through telnet and harbor HTTP is handled in there, so that we do not further detail this example here.

Seeking tracks. Every source has a method `seek` which takes as argument a number of second and goes forward this number of seconds. It returns the number of seconds effectively seeked: it might happen that the source cannot be seeked (if it is a live stream for instance), in which case the function will always return 0, or we might not be able to seek the given amount of time if we are near an end of track. For instance, the following script will seek forward 5 seconds every 3 second:

```
set("decoder.priorities.mad",100)
s = playlist("~/Music")
```

```
thread.run(every=3., {ignore(s.seek(5.))})
output(s)
```

The first line instructs to use the mad decoder for MP3 files, because it has better support for seeking than the default decoder FFmpeg. It is possible to give a negative argument to `seek`, in which case it will try to seek backward in time.

End of tracks. The operator `source.on_end` can be used to call a function some time before the end of each track. In addition to the `delay` parameter, which specifies this amount of time, the operator takes the source whose tracks are to be processed and a handler function, which is executed when a track is about to end. This handler function takes as arguments, the amount of remaining time and the metadata for the track. For instance, the following track will say the title of each song 10 seconds before it ends:

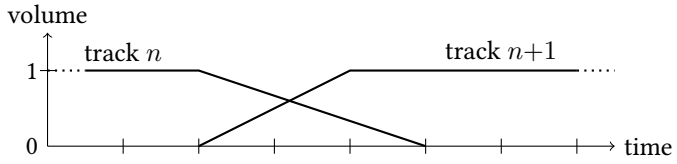
```
s = playlist("~/Music")
q = request.queue()
def speaker(t, m)
  title = m["title"]
  q.push.uri("say:Last song was #{title}")
end
s = source.on_end(delay=10., s, speaker)
s = add([q, s])
output(s)
```

You should now recognize a usual programming pattern. The main source is `s`, which is added to a queue `q`. We use `source.on_end` to register the handler function `speaker`, which inserts into the queue a request to say the title.

The operator `source.on_end` is also used behind the curtains to implement `fade.out`.

6.4 Transitions

So far, we have seen that we can easily play several music files sequentially (for instance with the `playlist` operator) or switch between two sources (using the `fallback` or `switch` operators). However, the resulting transitions between two tracks are quite abrupt: one track ends and the other starts. We often want crossfading transitions between tracks, which means that the volume of the first track should be progressively lowered and the one of the second progressively increased, in such a way that we hear the two during the transition:



Note that, as figured in the graph above, we don't necessarily want the duration of the transition to be the same for all tracks: for instance, the transition should be shorter (or there should even be no transition) for tracks starting or ending abruptly.

Liquidsoap supports both

- transitions between two distinct sources (for instance, when changing the source selected by a `switch`), and

- transitions between consecutive tracks of the same source.

The latter are more tricky to handle, 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 we will run into synchronization issues.

Cue points. Before discussing transitions, we should first ensure that our tracks begin and end at the right time: some songs features long introductions or long endings, that we would like not to play on a radio (think of a Pink Floyd song). In order to do so, we would rather avoid directly editing the music files, and simply add metadata indicating the time at which we should begin and stop playing the files: these are commonly referred to as the *cue in* and the *cue out* points.

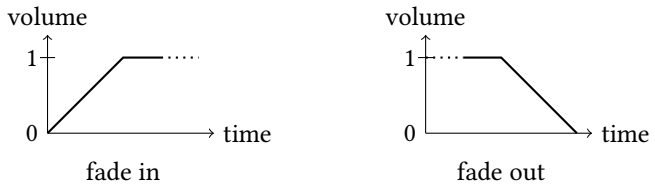
The `cue_cut` operator takes a source and cuts each track according to the cue points which are stored in the metadata. By default, the metadata `liq_cue_in` and `liq_cue_out` are used for cue in and cue out points (the name of the metadata can be changed with the `cue_in_metadata` and `cue_out_metadata` parameters of `cue_cut`), and are supposed to be specified in seconds relative to the beginning of the track. Negative cue points are ignored and if the cue out point is earlier than the cue in point then only the cue in point is kept.

For instance, in the following example, we use the `prefix` argument of `playlist` to set `liq_cue_in` to `3` and `liq_cue_out` to `9.5` for every track of the playlist:

```
s = playlist(prefix="annotate:liq_cue_in=3,liq_cue_out=9.5:", "~/Music")
s = cue_cut(s)
output(s)
```

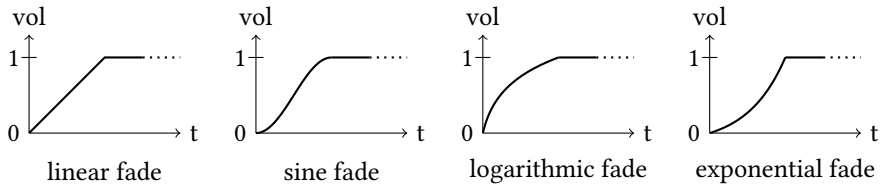
We will thus play every song of the playlist for 6.5 seconds, starting at second 3. In practice, the metadata for cue points would either be hardcoded in the files or added for each file with `annotate` in the playlist. In a more elaborate setup, a `request.dynamic` setup would typically also use `annotate` in order to indicate the cue points, which would be fetched by your own scheduling back-end (for instance, the cue points could be stored in a database containing all the songs).

Fading. In order to have smoother transitions, a first way to proceed is to progressively increase the volume from 0 (the minimum) to 1 (the maximum) at the beginning of tracks and progressively decrease the volume from 1 to 0 at the end of tracks: these operations are respectively called *fading in* and *fading out* and their effect on the volume can be pictured as follows:



The operators `fade.in` and `fade.out` respectively fade in and out every track of the source given as argument. The `duration` parameter controls the duration in seconds of the fade: this corresponds to the length of the ramp on the above figures, by default it takes 3 seconds to entirely change the volume. The duration can also be changed by using setting the metadata `liq_fade_in` (the name can be changed by with the `override_duration` parameter of the functions). Finally, the parameter `type` controls the shape of the fade: it can respectively

be "lin", "sin", "log" and "exp" which will respectively change the shape of the fade as follows:



The default shape is linear (it is the simplest), but the sine fade tends to be the smoother for the ear. For instance, the script

```
s = fade.in(duration=4., type="sin", s)
s = fade.out(s)
```

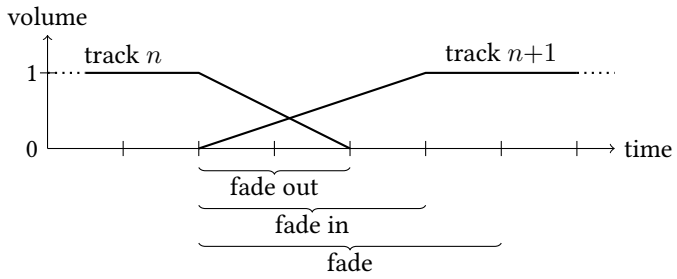
will add a sinusoidal fade in of 4 seconds and a linear fade out of 3 seconds to every track of the source s.

Transitions between successive tracks. In order to have nice transitions between successive tracks of a source, the simplest way is to use the crossfade operator which takes a source and performs fading: it fades out the current track, fades in the next track and superposes the two. In order to crossfade the tracks of a source s you can simply write

```
s = crossfade(s)
```

et voilà!

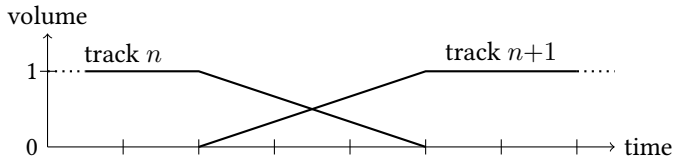
The fade_out, fade_in and duration parameters of crossfade control the length in seconds of the fade out, fade in and the total duration of the transition as figured below:



In this example, we have a fade out time of 2 seconds, a fade in time of 3 seconds and a fade duration of 4 seconds, which corresponds to the following script:

```
s = crossfade(fade_out=2., fade_in=3., duration=4., s)
```

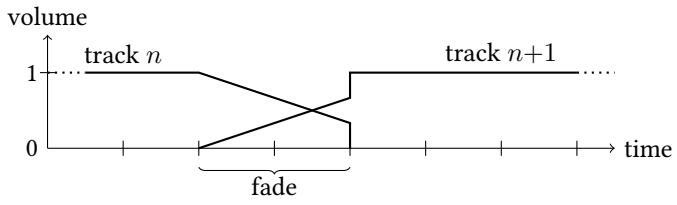
The default values are 3 seconds for fade in and out and 5 seconds for fade:



The total duration should always be strictly longer than the one of the fades, otherwise the fades will not be complete and you will hear abrupt changes in the volume. For instance, with a fade in and out of 3 seconds and a fade duration of 2 seconds

```
s = crossfade(fade_out=3., fade_in=3., duration=2., s)
```

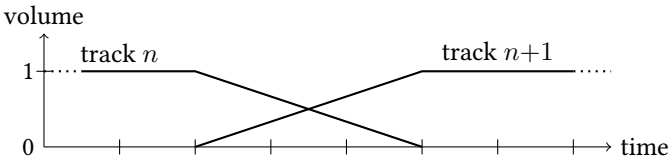
we will have the following incomplete transitions:



The fade duration can be changed by setting the `liq_cross_duration` metadata (and the name of the metadata can be changed in the `override_duration` parameter).

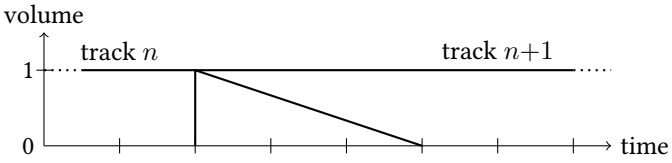
Smart crossfading. The `crossfade` operator has a "secret" option called `smart` to produce more relevant transitions. When you set this parameter to `true`, Liquidsoap inspects the relative loudness of the ending track and the starting track and applies a different transition depending on their values. The general idea is that we want to fade music which is soft (i.e. not very loud) and apply no fading to more brutal (i.e. loud) music. In details the transitions are as follows:

- if both tracks are not very loud (both are below the value specified by the `medium` parameter) and the loudness are comparable (their difference is below the value specified by the `margin` parameter), we apply a regular transition:



- if both tracks are very loud (both are above the value specified by the `high` parameter), we apply no transition:

- if the first track is not loud and the second one is, we only fade the first



and dually if the first one is loud and the second one is not.

Under the hood: the `cross` *operator.* In the case you want to customize transitions between tracks, you should use the `cross` operator which allows the fully specify which transition we should apply. In fact, the `crossfade` operator is itself programmed in the standard library using the `cross` operator. This operator takes as argument `duration` the duration of the fade (whose default value is 5 seconds), a function specifying the transition, and the source whose tracks should be crossfaded. The type of the transition function is

```
({metadata : [string * string], db_level : float, source : source},
 {metadata : [string * string], db_level : float, source : source})
-> source
```

It takes two arguments, respectively corresponding to the end of the old track and the beginning of the new track, which consist in records providing, for both of them,

- the metadata,
- the loudness in dB,
- the source corresponding to the track,

and returns a source corresponding to the crossfading of the two tracks. For instance, the usual fading can be achieved with

```
def f(a, b)
  add(normalize=false, [fade.out(a.source), fade.in(b.source)])
end
s = cross(f, s)
```

The transition function `f` simply adds the source of the ending track, which is faded out, together with the source of the beginning track, which is faded in. It is important that we set here the `normalize` argument of the `add` to `false`: otherwise the overall volume will be divided by two (because there are two sources) and we will hear a volume jump once the transition is over.

Let us give some other examples of transition functions. If we want to have no transition, we can use the `sequence` operator to play the end of the old source and then the beginning of the new source:

```
def f(a, b)
  sequence([a.source, b.source])
end
```

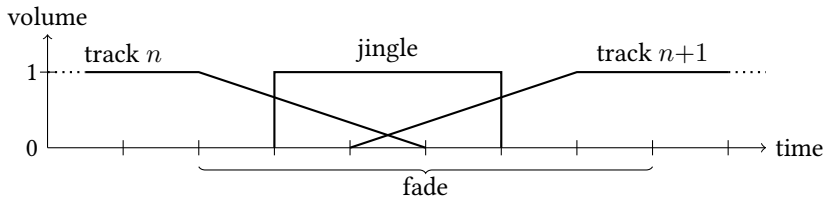
We could take this opportunity to insert a jingle in between the tracks:

```
def f(a, b)
  sequence([a.source, once(jingle), b.source])
end
```

(we suppose there that `jingle` is a playlist of jingles). Or we could to all the three at once. Namely, suppose that we have a jingle and we want to achieve the following:

- we begin by fading out `a` for 3 seconds,
- after 1 second, we start playing the jingle (say that it lasts for 3 seconds),
- after 2 seconds, we start playing `b` which is faded in.

Graphically, this would look like the following:



This can be achieved by the following function which is basically adding the old source faded out, the jingle and the new source faded in:

```
def f(a, b)
  add(normalize=false, [
    fade.out(a.source),
    sequence([blank(duration=1.), once(jingle)]),
    sequence([blank(duration=2.), fade.in(b.source)]),
  ])
end
```

The operators `sequence` are used here to add some blank at the beginning of the source in order to delay the moment where they are started. The "real" duration of the fade is 5 seconds, but we set the `duration` parameter to 6 to have 1 extra second for safety. In order to illustrate the use of the `metadata` fields, suppose that we want to have no transition, but want to insert a jingle when the metadata `jingle` of the new track is set to `true`. This can be achieved with:

```
def f(a, b)
  if b.metadata["jingle"] == "true" then
    sequence([a.source, once(jingle), b.source])
  else
    sequence([a.source, b.source])
  end
end
```

Here, we make use of the field `metadata` of `b` which contains the metadata for the starting track.

Finally, in the case where the current track ends unexpectedly, we might not have enough time to perform the transition. For instance, when we skip the current track of a source, we immediately go to the next track. The `minimum` parameter of `cross` controls how much time in advance we should have to perform the transition: if the remaining time of the current track is below this value, we simply don't apply any transition.

Transitions between different sources. The operators which allow switching between different sources (`switch`, `fallback`, `rotate` and `random`) also allow specifying the transitions to be applied when switching from one source to the other. A *transition* is described by a function taking two sources as arguments and returning a new source: the first argument is the source which is about to be left, the second argument is the newly selected source, and the returned source is the result of their combination. This is similar to the transitions for `cross`, excepting that we don't have the power and metadata in this case. The default transition is the function

```
fun (a, b) -> b
```

which simply discards the stream from the old source and returns the one of the new one. In practice, the first argument is often irrelevant because Liquidsoap cannot predict accurately when the next switch will occur.

The switching operators all take an argument `transition_length` which controls the length of the transition in seconds, i.e. how long the two sources `a` and `b` will overlap, the result of the overlap being computed by the transition function. This transition duration can be overridden by passing the metadata named `liq_transition_length` (the name of this metadata can be changed with the `override` parameter). The operators also take a list `transitions`: the nth element of this list is the transition function that will be used when switching to the nth source.

In order to illustrate this, suppose that we have two sources: `live` which is a live source available from time to time (for instance, a DJ connecting to an `input.harbor` source) and `music` which is a local music source (for instance, a playlist). In such a situation, we would define with a fallback which plays the live source if available and defaults to the music source otherwise:

```
radio = fallback(track_sensitive=false, [live, music])
```

We want to enhance our setup and have transitions such that

- when we switch to the live source, we want to hear "And now the live show!" while the sound of the live source progressively fades in,
- when we switch back to the music source, we want to hear a jingle and then the music source.

This will be achieved by implementing two dedicated transition functions:

- the first one, `to_live(a, b)`, will add a request to say the text once with the source `b` (which contains the beginning of the live show),
- the second one, `to_music(a, b)`, uses the operator `sequence` to play a jingle and then the source `b`.

Their code is given below:

```
def to_live(a, b)
  add(normalize=false, [
      once(single("say:And now the live show!")),
      fade.in(duration=4., type="sin", b)
    ])
end
def to_music(a, b)
  sequence([single("jingle.mp3"), b])
end
radio = fallback(track_sensitive=false, transition_length=5.,
                 transitions=[to_live, to_music], [live, music])
```

We finally use the `fallback` operator to play the live source if available and the music source otherwise. We set the list `transitions` of transitions to `to_live` and `to_music`, which are respectively used when switching to the `live` and `music` sources.

Smooth add. We have seen earlier that we could insert a jingle every 30 minutes in a stream with by adding to the main `music` source a track of `jingle` as follows:

```
jingles = delay(1800., playlist("jingles"))
music = playlist("~/Music")
radio = add([jingles, music])
output(radio)
```

If we want to fade the jingle instead of adding it abruptly when available, we could use the above functions to program our fades. But fortunately, this is already programmed for us in the standard library, with the `smooth_add` function. It takes as argument a `special` source which is available from time to time (the jingles in our case), a `normal` source which is usually available (the music source in our case) and, whenever a track of `special` is available adds it on top of `normal` with a faded transition. The argument `duration` controls the duration of the transition (1 second by default) and the parameter `p` controls the proportion of the normal source in the mix (0.2 by default with means 20% normal source and 80% special source). In the above script, we could use it by replacing the penultimate line by

```
radio = smooth_add(duration=0.5, special=jingles, normal=music)
```

6.5 Signal processing

Now that we have seen the ways of generating sound, we should see ways to shape the sound.

Amplification. The first basic sound effect is *amplification*, i.e. raising or lowering the volume of a source. This is basically achieved with the `amplify` operator which takes a float coefficient and a source, and amplifies the sources by the coefficient. For instance, we can halve the loudness of a source `s` by

```
s = amplify(0.5, s)
```

As for most parameters of audio effects, `amplify` also accepts getters as coefficients, which allow modifying the value dynamically. For instance, we could use an interactive variable for the amplification parameter:

```
a = interactive.float("main_volume", 1.)
s = amplify(a, s)
```

this would allow changing the value of the amplification on the telnet using the command

```
set main_volume 1.2
```

We could also fetch the value of the volume from the contents of the `volume` file as follows:

```
a = file.getter.float("volume")
s = amplify(a, s)
```

The `file.getter.float` regularly looks at the contents of the file `volume` and returns an updated value. Such mechanisms could be handy to provide the user with a way to adjust his volume for a live show using a graphical interface.

The `amplify` parameter also support setting the amplification coefficient using metadata: if the metadata `liq_amplify` is specified then its value will be taken as coefficient for current track (the name of the metadata itself can be changed with the `override` parameter `amplify`). This value can either be specified as a float coefficient (such as `0.7`) or in decibels (such as `-3.10 dB`).

ReplayGain. In particular, `amplify` is quite useful if you want to have all your audio files playing at the same loudness, without having to re-encode them: we can simply amplify each track differently based on a metadata in order to have more homogeneous loudness. There is a standard way of computing the required amplification factor, called *ReplayGain*, which takes in account the human perception of the sound in order to suggest a volume correction, thus ensuring a comfortable, consistent listening experience. Many tools are available to precompute this value and store it as a metadata in the file, for instance

```
loudgain -s i file.mp3
```

will add the `replaygain_track_gain` metadata to `file.mp3` and set it to a value such as `-1.69 dB` indicating that we should lower the volume by 1.69 decibels (which amounts to performing an amplification by 0.82) in order to reach a standard volume. If we assume that our files are tagged in this (standard) way, we can use the `amplify` operator to apply the gain correction as follows:

```
s = playlist("~/Music")
s = amplify(override="replaygain_track_gain", 1., s)
```

For convenience, the amplification by this metadata is defined in the standard library as the `replaygain` operator, so that we can even simply write

```
s = replaygain(playlist("~/Music"))
```

If not all your files are tagged with ReplayGain metadata you can use the command

```
enable_replaygain_metadata()
```

to instruct Liquidsoap to compute it for every played file: for each file, it will run a script (called `extract-replaygain`) which will try to look if the ReplayGain metadata is present, and if not will compute it (using `ffmpeg`). If you want instead to perform it on a per-file basis, you can use the protocol `replaygain:` which instructs to compute the ReplayGain of a file, with the same method. For instance,

```
s = replaygain(single("replaygain:test.mp3"))
```

will play the file `test.mp3`, computing its ReplayGain beforehand, and correcting its volume. Incidentally, we recall that the `prefix` parameter of playlists can be used add this protocol to all the files in the playlist:

```
s = replaygain(playlist(prefix="replaygain:", "~/Music"))
```

The operation of computing the ReplayGain for a given file is a bit costly so that we strongly advice to perform it once for all for your music files instead of using the above mechanisms.

Normalization. The above ReplayGain allows performing *volume normalization* when playing music files: we want the loudness to be more or less constant during the stream. For files this is "easy" because we can compute this information in advance, however for live streams we have to proceed differently since we do not have access to the whole stream beforehand. For such situations, the `normalize` operator can be used: it continuously computes the loudness of a stream and adjusts the volume in order to reach a target loudness: this operation is sometimes called *automatic gain control*. Basically, the sound of a source `s` can be normalized with

```
s = normalize(s)
```

Of course, the `normalize` operator takes quite a number of optional parameters in order to control the way the normalization is performed:

- `target` is the loudness we are trying to achieve for the stream (-13 dB by default).
- `up` and `down` respectively control the time it takes to increase the volume when the stream is not loud enough and the time it takes to decrease the volume when it is too loud (the higher, the more times it takes). Default values are respectively 10. and 0.1: we want to quickly lower the volume because a sound too loud is irritating (or worse, can damage the user's ear), but we increase it more slowly in order not to be too sensitive to local variations of the loudness.
- `gain_min` and `gain_max` is the minimum and maximum gain we can apply (respectively -12 dB and 12 dB by default, which correspond to an amplification by 0.25 and 4): the second is particularly useful so that we do not amplify like crazy when the stream is almost silent.
- `threshold` controls the level below which we consider that we have silence and do not try to increase the volume anymore.
- `window` is the duration of the past stream we take in account as the current volume: default value is 0.5 seconds. Increasing this will make the operator less sensitive to local variations of the sound, but also less reactive.
- `lookahead` specifies how many seconds of sound we look at in advance (beware that this will introduce that amount of delay in the output stream).

For instance, in the script

```
s = playlist("~/Music")
s = normalize(window=4., down=.5, lookahead=2., debug=.5, s)
```

we use `normalize` with a window of 4 seconds, which is quite long in order not to be too sensitive to local variations of sound, and a time of 0.5 seconds to lower the volume. This results in quite a smooth volume variation, but which tends to be late: because of the window, we only realize that the sound has gone up quite some time after it has actually gone up. In order to compensate this, we use a lookahead of 2 seconds, which makes the normalization base its decisions on audio 2 seconds in the future. In this way, when the loudness suddenly goes up, the `normalize` operator will "guess" it in advance and begin lowering the volume before the peak occurs.

Tweaking the parameters can take quite some time. In order, to ease this, the `debug` parameter, when set, will print internal parameters of the normalization (the current loudness, the target loudness, the current gain, and the gain required to reach target loudness) at the specified interval in the logs. The messages are printed as debug message, so that you should ensure that you set the log level high enough to see them:

```
set("log.level", 5)
s = normalize(debug=.5, s)
```

If you need more custom logging the source exports methods to retrieve the current loudness (`rms`), volume (`gain`) and volume required to reach the target loudness (`target_gain`); those are given in linear scale and can be converted to decibels with `dB_of_lin`. For instance, the script

```
s = normalize(s)
s = source.run(s, every=1., {print("r: #{s.rms()} g: #{s.gain()}")})
```

prints the loudness and volume every second.

Finally, if you need more customization over the operator, you can have a look at its code and modify it. It is written in Liquidsoap, in the `sound.liq` file of the standard library.

Handling silence. Another basic signal-processing feature that everyone wants to have is *blank detection*. We want at all cost to avoid silence being streamed on our radio, because listeners generally don't like it and will go away.

Skipping blank. Itt might happen that some music file is badly encoded and features a long period of blank at the end. In this case we want to skip the end of the song, and this is precisely the role of the `blank.skip` function:

```
s = blank.skip(max_blank=2., s)
```

The parameter `max_blank` specifies how long we want to take before skipping blank, 2 seconds in our example. Other interesting parameters are `threshold`, the loudness in dB below which we consider sound as silence, and `min_noise`, the minimum duration of non-silence required to consider the sound as non-silent (increasing this value allows considering as blank, silence with some intermittent short noises).

Stripping blank. It might also happen that the DJ has turned his microphone off but forgotten to disconnect, in which case we want to get back to the default radio stream. To handle such situations, we cannot use `blank.skip` because we cannot skip a live stream, but we can resort to the `blank.strip` operator which makes a source unavailable when it is steaming blank, and is typically used in conjunction with fallback:

```
mic   = buffer(input.alsa())
mic   = blank.strip(max_blank=2., min_noise=.1, threshold=-20., mic)
music = playlist("~/Music")
radio = fallback(track_sensitive=false, [mic, music])
```

The `max_blank` parameter states that we wait for 1 second of silence before making the source unavailable and the `threshold` parameter means that we consider anything below -20 dB as silence. The `min_noise` parameter means that we require that there is 0.1 s of noise before making the source available again, so that we still consider as silent a microphone where there is no sound most of the time, excepting very short noises from time to time (such as a person walking around).

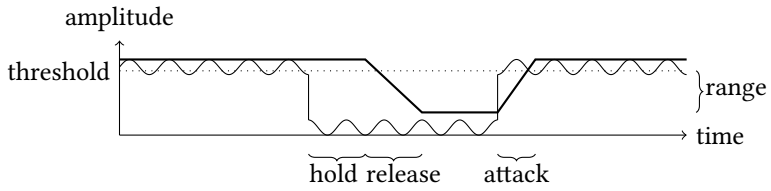
Gating. Sometimes we do want to stream silence, and when we do so, we want to stream real silence. When you have an input such as a microphone, it generally induces a small noise, which is almost unnoticeable when someone is talking, but quite annoying when we hear only that. In order to address this, the `gate` operator can be used to lower the volume when a source is almost silent. For instance,

```
mic = buffer(input.pulseaudio())
mic = gate(threshold=-30., range=-80., mic)
```

The useful parameters of gate are

- `threshold`: the level (in dB) from which we consider that sound is not silence anymore,
- `range`: by how much (in dB) we should lower the volume on "silence",
- `attack` and `release`: the time (in ms) it takes to increase or decrease the volume,
- `hold`: how much time (in ms) to wait before starting to lower the volume when we did not hear loud sounds.

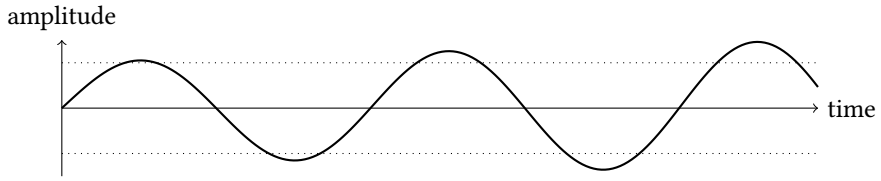
The following pictures both the amplitude of the sound (the squiggling curve) and the answer of the gate (the thick curve):



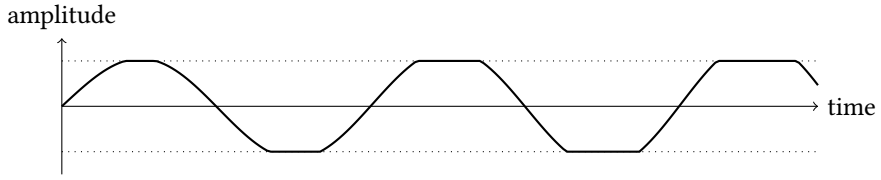
The internal state of the operator can be observed by the exported method `gate` which provides a value between 0 and 1 indicating whether the gate is "closed" or "open", i.e. if the volume is currently lowered or not.

Sound shaping. Now that we have presented the basic effects, which mainly operate on the volume of the sound, we will now be presenting some more advanced audio effects, which can be used to make the listening experience more homogeneous and give a "unique color" to your radio. We however need to begin by explaining one of the classical issues we have to face when operating on sound: clipping.

Clipping. We have indicated that all the audio samples should have a value between -1 and 1 in Liquidsoap. However, if we increase too much the volume of a source be it manually with `amplify` or in an automated way with `normalize`, it might happen that we obtain samples above 1 or below -1:



This is not a problem per se: Liquidsoap is perfectly able to handle such values for the samples. However, when we send such a signal to an output, it will be *clipped*: all the values above 1 will be changed to 1 and all the values below -1 will be changed to -1, in order to conform to the standard range for samples. Our signal will then look like this



As you can see, this operation is not particularly subtle and, as a matter of fact it has quite a bad effect on sound. If you want to test it you can try the script

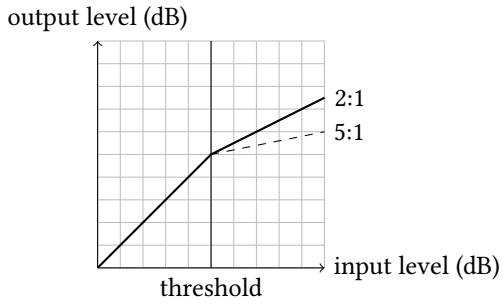
```
s = sine(440.)
s = amplify(s.time, s)
output(s)
```

It plays a sine, whose amplitude is given by the internal time of the source (`s.time()` is the number of seconds we have played of source `s`). On the first second, the amplitude will raise from 0 to 1 so that we will hear no distortion. After second 1, the amplification coefficient

will be greater than 1, so that clipping will occur, and it occurs more and more as time goes by. You should hear that our sine quickly does not sound like a sine anymore: we can hear high harmonics and the sound gets closer to the one of a square wave (try the `square` operator if you have never heard one). We will see next section operators which are useful to mitigate those effects.

In passing, if you insist of performing clipping in Liquidsoap (i.e. restricting samples between -1 and 1 as described above) you can use the `clip` operator.

Compressing and limiting. In order be able to increase the loudness of the signal without the sound artifacts due to clipping, people typically use *compressors* which are audio effects, which leave the signal untouched when it is not very loud, and progressively lowers when it gets loud. Graphically, the output level given the input level is represented in the following figure:

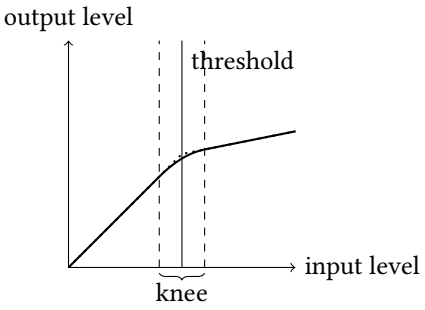


We can observe that below the *threshold* the output is the same as the input (the curve on the bottom left is a diagonal), and that above the threshold, the output increases more slowly than the input, following a *ratio* which can be configured (we have pictured ratios 2:1 and 5:1 to illustrate this). This operator is implemented in Liquidsoap as the `compress` operator. For instance, the compression of a source `s` above -10 dB at a ratio of 5:1 can be achieved with

```
s = compress(threshold=-10., ratio=5., s)
```

This operator has a number of useful parameters:

- the `threshold` (in dB) above which we should start compressing,
- the `ratio` of compression (in input dB per output dB, as illustrated above),
- the amplification by `pre_gain` and `gain` of the signal, respectively before and after compressing,
- the `attack` and `realease` time (in ms), which are the time it takes to respectively lower and increase the volume (those operations are generally not performed instantly in order to smoothen the variations and make them more natural),
- the `knee` (in dB) controls the width of the smoothing around the threshold:



- the `window` (in ms) used to smoothen the computation of the input level,
- the `lookahead` (in ms), i.e. how much time we look at the signal in the future (this is similar to what we already saw for `normalize`).

Finally, the `wet` parameter gives the original signal when set to 0 and the compressed signal when set to 1, and can be useful when configuring compressors, in order to evaluate the effect of a compressor on a signal.

A typical compressor, will have a ratio from 2 to 5: it will smoothly attenuate high loudness signals, which allows boosting its loudness while avoiding the clipping effect. Compressors with higher ratios such as 20 are called *limiters*: they are here to ensure that the signal will not get out of bounds. The standard library defines `limit`, which is basically a compressor with a ratio of 20, and can be thought of as a better version of `clip` with much less sound distortion induced on the sound. For instance, if we get back to the example we used to illustrate the effects of clipping, you can hear that the stream sound much more like a sine if we limit it, even when the amplification is higher than 1:

```
s = sine(440.)
s = amplify(s.time, s)
s = limit(s)
output(s)
```

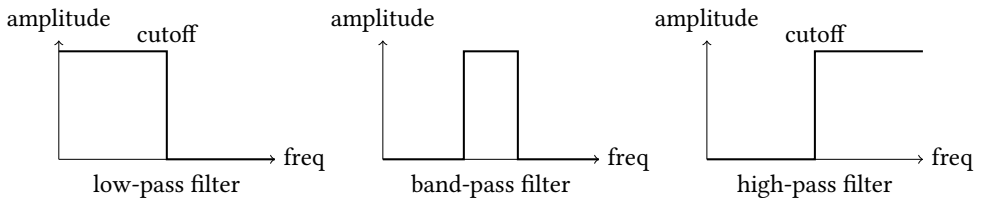
Since the `normalize` function is not perfect, it might happen that it produces a very loud sound for a short period of time. In order to avoid the clipping it would induce, it is advisable to always pipe the output of `normalize` to a limiter:

```
s = limit(normalize(s))
```

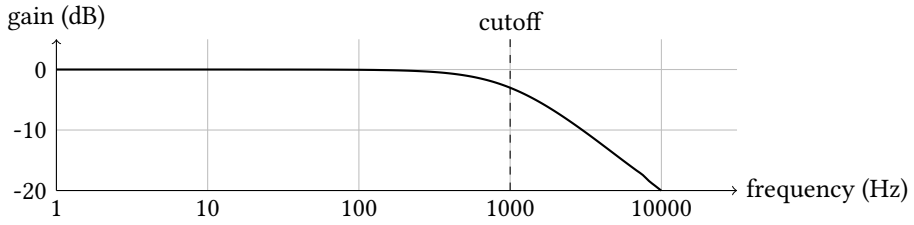
Filters. The last basic effect we are going to introduce are *filters*, which only keep some frequencies of the input signal. There are three main types:

- low-pass filters only keep low frequencies (below a given *cutoff* frequency),
- high-pass filters only keep high frequencies (above a given *cutoff* frequency),
- band-pass filters only keep frequencies between two boundaries.

Ideal filters would have a frequency response as indicated in the following figures:



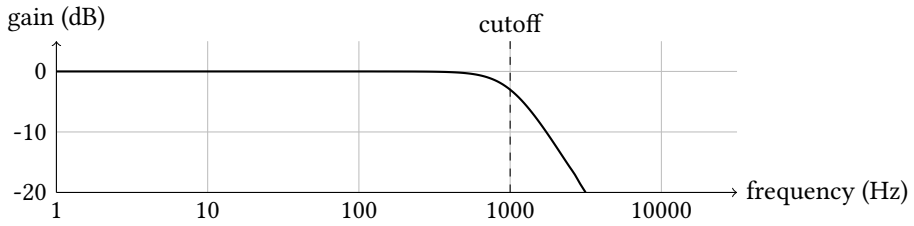
For instance, a low-pass filter would keep all the frequencies below the cutoff exactly as they were in the original signal, and keep none above. In practice, the transition between kept and removed frequencies is smother and the actual filter response of a low-pass filter is rather like this:



(to be precise, this is the response of a first-order low pass filter, as implemented in the `filter.rc` operator, with a cutoff frequency of 1000 Hz). Designing filters is an art rather than a science, and this is not the place to explain it. We will simply mention here that, in practice, using biquadratic filters is generally a good idea, because they offer good balance between efficiency and precision. Those are implemented as `filter.iir.eq.*` operators such as

- `filter.iir.eq.low`: a biquadratic low-pass filter,
- `filter.iir.eq.high`: a biquadratic high-pass filter,
- `filter.iir.eq.low_high`: a band-pass filter obtained by chaining a low- and a high-pass biquadratic filter.

For comparison, the frequency response of `filter.iir.eq.low` with a cutoff frequency of 1000 Hz is



You can observe that it is much sharper than the first-order one, and thus closer to the ideal filter. The functions `filter.rc` and `filter` are computationally cheaper (in particular the former) but are of lower quality (in particular the former): the change of amplitude around the cutoff frequency is less abrupt for `filter.rc` than for biquadratic filters and `filter` does not handle well high cutoff frequencies (higher than the quarter of the sampling rate, which often means around 10 kHz). The `filter.iir.butterworth.*` filter are of good quality (and their quality can be arbitrarily increased by increasing their `order` parameter), but require

more computations (and the higher the order is, the more CPU is used).

A typical use of filters is (obviously) to remove unwanted frequencies. For instance, cheap microphones, often produce noise at low frequencies, which can be removed using a high-pass filter. This is why the standard library defines the function

```
def mic_filter(s)
  filter(freq=200.,q=1.,mode="high",s)
end
```

which can be used to perform such a cleaning of the sound by removing frequencies below 200 Hz.

Another typical use of filters is to increase the frequencies we like. For instance, increasing bass frequencies makes the sound warmer and thus more pleasant to listen for background music. If we want to do so, we can extract the low frequencies (say, below 200 Hz) from a source `s` using a low-pass filter, amplify them (say, by 6 dB) and add them back to the original sound:

```
b = amplify(lin_of_dB(6.), filter.iir.eq.low(frequency=200., s))
s = add([s, b])
```

However, if we do things in this way, the risk is high that we are going to clip and thus hear saturation from the basses. As explained above, a much more pleasant solution consists in using a limiter after increasing the volume. In this way, we can handle a 8 dB increase of the frequencies below 200 Hz without any problem:

```
b = limit(pre_gain=8., filter.iir.eq.low(frequency=200., s))
s = add([s, b])
```

This is implemented in the standard library as the `bass_boost` operator, so that the above can be more concisely written

```
s = bass_boost(frequency=200., gain=8., s)
```

Multiband compression. We are now ready to introduce the effect that you were all waiting for, which is in fact a combination of previous effects: *multiband* compression aka the *big fat FM radio sound.* This is what is used in most commercial music radios so that, when you listen to songs in your car without thinking too much, you are not disturbed by changes in the dynamics of songs. Whether you like it or not, this can easily be done in Liquidsoap. This is basically achieved by splitting the signal in various bands of frequencies (using band-pass filters such as `filter.iir.eq.low_high`), independently compress each of those (using `compress`), and add them back together (using `add`). In other words, we apply the same principle as the above "bass booster" to all the spectrum.

For instance, the script

```
compress = compress(attack=100., release=200., threshold=-20.,
                    ratio=6., gain=7.)
s = add(normalize=false, [
  compress(filter.iir.eq.low     (frequency=200.,        s)),
  compress(filter.iir.eq.low_high(low=200., high=800.,   s)),
  compress(filter.iir.eq.low_high(low=800., high=1500.,  s)),
  compress(filter.iir.eq.low_high(low=1500., high=8000., s)),
  compress(filter.iir.eq.high    (frequency=8000.,       s))
])
```

defines a `compress` function by specifying values for some of the arguments of the original one. It then splits the sound in 5 bands: below 200 Hz, 200 to 800 Hz, 800 to 1500 Hz, 1500 to 8000 Hz and above 8000 Hz. Finally, it applies compression to each of these bands and adds back the bands.

For convenience, the function `compress.multiband` of the standard library already implements this: it takes in account the specification of the bands, which consists in its frequency (this is the higher frequency, the lower one is taken from the previous band) as well as the threshold, ratio, attack and release time parameters of the corresponding compressor. The script

```
s = compress.multiband(s, [
  {frequency=200.,   attack=100., release=200.,
   threshold=-10.,   ratio=4.,    gain=10.},
  {frequency=800.,   attack=100., release=200.,
   threshold=-10.,   ratio=4.,    gain=6.},
  {frequency=1500.,  attack=100., release=200.,
   threshold=-8.,    ratio=4.,    gain=6.},
  {frequency=8000.,  attack=100., release=200.,
   threshold=-6.,    ratio=4.,    gain=6.},
  {frequency=40000., attack=100., release=200.,
   threshold=-4.,    ratio=4.,    gain=6.}
])
```

is therefore roughly the same as the above one, excepting that we are varying the parameters for fun. Of course getting the parameters right requires quite some trial and error, and listening to the results. Below, we describe `compress.multiband.interactive` which helps much by providing a simple graphical interface in which we can have access to those parameters.

Other effects. We have presented the most useful effects above, but some others are available in Liquidsoap. You are advised to have a look at the documentation to discover them.

In particular, there are some effects on stereo stream such as left/right *panning* (the `pan` operator) or modifying the width of the signal (the `width` operator, which takes a float parameter such that -1 turns the original source into mono, 0 returns the original source and a value greater than 0 returns the source with "widened" sound). There are some traditional effects used in music such as `echo` or `flanger`. And finally, there are some effects which operate on the pitch of the sound such as `stretch` which reads a source quicker than realtime thus resulting in high-pitched sounds and the `soundtouch` operator which is able to perform *pitch-shifting* and *time-stretching*.

LADSPA and LV2 plugins. In the case where you are not satisfied by the builtin operators, Liquidsoap support LADSPA and LV2 plugins. Those are standards for signal processing plugins, for use in any application. Many free plugin packs are available, among which we recommend

- Calf Studio Gear,
- Linux Studio Plugins,
- Steve Harris' plugins,
- Tom's plugins.

Once installed on your system, those plugins will appear has operators named `ladspa.*` or `lv2.*` (here `*` is replaced by the plugin name). You can use the command

```
liquidsoap --list-functions
```

to list all Liquidsoap operators and thus discover those which are present in your installation. You can then use the `-h` option to read the help about a particular plugin. For instance, if we type

```
liquidsoap -h ladspa.fastlookaheadlimiter
```

we obtain

```
Fast Lookahead limiter by Steve Harris.

Type: (?id : string, ?input_gain : {float}, ?limit : {float}, ?release_time :
↪  {float}, source(audio=pcm(stereo), video='a, midi='b)) ->
↪  source(audio=pcm(stereo), video='c, midi='d)

Category: Source / Sound Processing
Flag: extra

Parameters:

 * id : string (default: "")
     Force the value of the source ID.

 * input_gain : {float} (default: 0.)
     Input gain (dB) (-20 <= `input_gain` <= 20).

 * limit : {float} (default: 0.)
     Limit (dB) (-20 <= `limit` <= 0).

 * release_time : {float} (default: 0.507499992847)
     Release time (s) (0.01 <= `release_time` <= 2).

 * (unlabeled) : source(audio=pcm(stereo), video='a, midi='b)
```

from which we understand that we have some sort of limiter, whose input gain, threshold and release time can be configured, and can use it in a script as follows:

```
s = ladspa.fastlookaheadlimiter(limit=-3., s)
```

Plugins can be a bit difficult to understand if you have no idea what the plugin does, in which case the documentation on the author's websites can be useful.

Many other plugins are provided by the FFmpeg library. They are presented later, in there, since they are a bit more difficult to use and you are most likely to use them for video.

Stereo Tool. A last possibility to handle your sound is to use software dedicated to producing high quality radio sound, such as *Stereo Tool*. If you want to do so, you can use the `pipe` operator which allow exchanging audio data with external software through the standard input and output and is detailed in a later section. Typically, one would use it to handle a source `s` with Stereo Tool as follows:

```
s = pipe(replay_delay=1.,
  process='/usr/bin/stereo_tool_cmd_64 - - -s myradio.sts -k "seckey"',
  s)
```

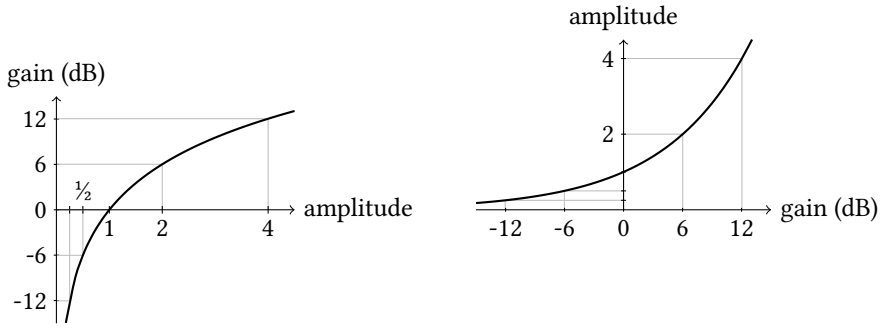
The `process` argument gives the program we want to run along with its options (here, you should replace `/usr/bin/stereo_tool_cmd_64` by the actual path where the Stereo Tool binary is located, `myradio.sts` by your configuration file and `seckey` by your actual license key).

Playing with parameters. We would now like to make a few remarks about the way parameters can be handled in order to configure sound effects. You will certainly experience that you quickly face with loads of parameters and, when you want to find the right values, it can be very tedious to change them in your script, save the file, and relaunch the whole script in order to listen to the effect.

Decibels. Before dealing with parameters themselves, we remind you that there are two ways to measure amplitude, either *linearly* or in *decibels*. The relationship between the two is not simple: recall from the beginning of the book that linear l and decibel d measurements are related by the relations $d=20\log_{10}(l)$ and $l=10^{d/20}$. What you should remember is that

- 0 dB is an amplitude of 1,
- subtracting 6 dB amounts to dividing the amplitude by 2,
- adding 6 dB amounts to multiplying the amplitude by 2.

Graphically, the relationship between the linear amplitude and the gain in decibels is pictured below in both ways:



In Liquidsoap, the functions `lin_of_dB` and `dB_of_lin` can be used to convert between the two: the first converts decibels in linear units and the second does the converse. For instance, we can amplify a source `s` by 4 dB with

```
s = amplify(lin_of_dB(4.), s)
```

When using operators, you should always check in the documentation the unit for the amplitudes. Unfortunately, both exist in nature (for instance, `amplify` takes a linear parameter whereas most effects such as compressors expect decibels).

Getters. Most sound operators take *getters* as arguments, as already explained in an earlier section. For instance, the type of `amplify` is (roughly)

```
({float}, source) -> source
```

The first argument, which corresponds to the amplification parameter, is of type `{float}` which means that both

- a `float`, or
- a function of type `() -> float`

are accepted. This means that we can either directly provide a value as in

```
s = amplify(0.5, s)
```

or we can provide a function which returns a `float` each time it is called, which can be different each time. For instance, in the script

```
t0 = time()
def a()
  time() - t0
end
s = amplify(a, s)
```

we store in `t0` the time (in seconds) at the startup of the program, define `a` as the function which returns the difference between the current time and the startup time (in seconds), and use it as amplification factor for the source `s`: this means that after running the script for n seconds our source `s` will be amplified by n. We also recall that this can be written more concisely as

```
t0 = time()
s  = amplify({time() - t0}, s)
```

and, in fact, we could even directly use the function `time.up` which gives the time since startup

```
s  = amplify(time.up, s)
```

The fact that we can use getters as parameters is quite useful for retrieving parameters from external sources. For instance, the function `file.getter.float` has type

```
(string) -> () -> float
```

It takes as argument a file name (a string) and returns a function which, each time it is called, returns the float which is contained in the file (and this is done in an efficient way). This means that the following script

```
a = file.getter.float("volume")
s = amplify(a, s)
```

will amplify the source `s` by the value indicated in the file `volume`: as soon as you change the value in this file, you will hear a corresponding change in the volume.

Interactive variables: telnet. Instead of using files to store parameters as described above, our preferred way of handling those is with *interactive variables*. These can be thought of as references whose contents can be changed in various ways. An interactive variable of type `float` can be created with the function `interactive.float` (and similarly interactive strings and booleans can be created by `interactive.string` and `interactive.bool`). This function takes as argument the name of the interactive variable, which we will use to modify it, as well as the initial value. For instance, we can amplify a source `s` by an interactive float named `main_volume` by

```
a = interactive.float("main_volume", 1.)
s = amplify(a, s)
```

A first way to modify such variables is through the telnet server. It can be started by adding

```
set("server.telnet", true)
```

add the beginning of the script. We can then connect to the telnet server by typing

```
telnet localhost 1234
```

Here, we suppose that we are on the machine running the script, otherwise `localhost` has to be replaced by its address. `1234` is the default port for the telnet server. Then, we can change the value of the interactive variable `main_volume` to 0.75 by typing

```
var.set main_volume = 0.75
```

to which the server will answer

```
Variable main_volume set.
END
```

We can also retrieve the value of the variable by typing

```
var.get main_volume
```

which will prompt the answer

```
0.75
END
```

The `telnet` command can also read commands to send from its standard input, allowing to automate the process of setting variables in scripts. For instance, we can type

```
echo "var.set main_volume = 0.5" | telnet localhost 1234
```

to set `main_volume` in one go.

Interactive variables: persistency. One issue with interactive variables is that they are not *persistent*: if we stop the script and run it again, their values are lost and we get the default values again. However, this is easily solved by using the `interactive.persistent` function, which takes a filename and stores the values of all interactive variables in this file (in JSON format, which should easily be readable). For instance, if you end the previous script with

```
interactive.persistent("script.params")
```

you will observe that a file `script.params` has been created and its contents is

```
[ { "main_volume": 0.5 }, [  ], [  ], [  ] ]
```

which, without entering the details, contains the value for `main_volume`. Moreover, it will automatically be read on next run of the script, so that interactive variables will keep their values across executions.

There is one important caveat: the function `interactive.persistent` must be called *after* all interactive values have been created (i.e. after all calls to functions `interactive.float` and similar), otherwise previous values are not taken in account when restarting scripts. If you do not want to think too much, follow this simple rule: put any call to `interactive.persistent` toward the end of the script!

Interactive variables: web interface. All this is very nice, but having to go through a telnet interface to change values is not very user-friendly. Fortunately, we can also get a web interface for free, simply by typing

```
interactive.harbor()
```

This will run a web server, which is accessible at the URL

```
http://localhost:8000/interactive
```

which can be configured by setting parameters of the function, and allows modifying the values of the variables. If you connect to it, you will see a webpage like

where we can edit in realtime the value of the interactive variable (of course if we had many variables we would have one line for each of them). If we specify the minimal and maximal value of the interactive variable (`min` and `max` parameters of `interactive.float`) we moreover get a slider, and if we moreover set the `description` it will be displayed. This means that by changing the declaration of the interactive variable to

```
a = interactive.float("main_volume", description="Our volume",
                      min=0., max=3., 1.)
```

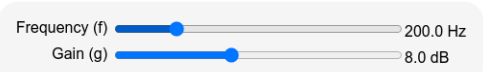
the webpage will change to

In this way you easily get access to a convenient interface for setting your parameters, and their values can be stored on the long run by using `interactive.persistent` as explained above.

In order to provide another illustration, suppose that we want to setup a bass booster for our radio source `s`. The way we are going to design it is by setting up two interactive variables `f` and `g` for the frequency and the gain

```
interactive.harbor()
f = interactive.float("f", description="Frequency", min=0., max=1000.,
                      unit="Hz", 200.)
g = interactive.float("g", description="Gain", min=0., max=20.,
                      unit="dB", 8.)
b = bass_boost(frequency=f, gain=g, s)
s = add([s, b])
output(s)
interactive.persistent("bb.params")
```

and tweaking them using the interactive variables webpage which looks like



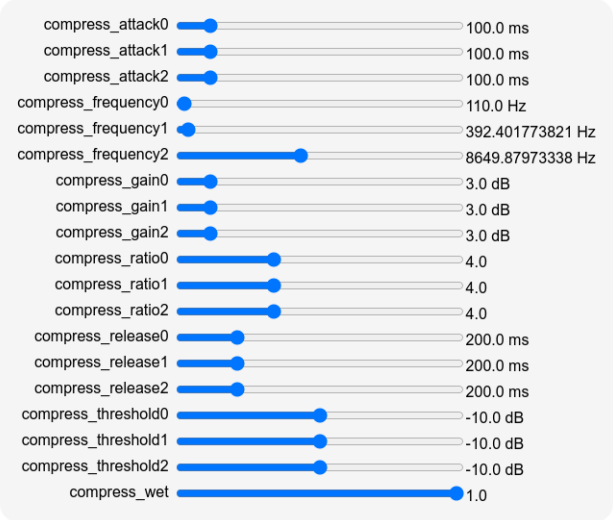
Once the right values found, they will be stored in the `bb.params` files, but you could then hardcode them in your script for more resiliency.

As a last example, suppose that we want to set up a multiband compressor. Well, we could do the same as above for the parameters of `compress.multiband`, but it becomes quite tedious to create interactive variables for all the parameters of the function, for each band. Fortunately, the `compress.multiband.interactive` operator can do this for us: we provide it with the

number of bands we want to have and it creates a `compress.multiband` instance as well as all the interactive variables for us. For instance, given a source `s`, the script

```
interactive.harbor()
s = compress.multiband.interactive(bands=3, s)
output(s)
interactive.persistent("comp.params")
```

will give rise to the following interface



which allows to easily set up the multiband compressor using our ears and our mouse. The *wet* parameter allows to compare the output with and without compression, as explained below.

Interactive variables: OSC. Another way to modify interactive variables is through the OSC (*Open Sound Control*) protocol, which is used to communicate values over a local network. There is plenty of software for your tablet or your phone, which emulate controllers with loads of sliders and send their values using this protocol. Each of the sliders has an OSC address, which looks like "`/the/address`", whose name depend on the software you use. When launching your software, you should first enter the IP address of the machine you want to communicate with (in our case, the machine where Liquidsoap is running) and the port on which we want to communicate (Liquidsoap uses 7777 by default, this can be changed by setting the `osc.port` parameter). The function `interactive.float` takes an `osc` parameter which can be used to specify an OSC controller to listen to: when set, the variable will change when the corresponding controller updates its value. For instance, the script

```
set("osc.port", 9000)
a = interactive.float(osc="/volume", "main_volume", 1.)
s = amplify(a, s)
output(s)
```

listens on the port 9000 for OSC events and changes the value of the interactive variable `a` when a new float value is sent at the OSC address "`/volume`".

In passing, Liquidsoap also offers some more low-level functions in order to manipulate OSC values:

- `osc.float` takes an OSC address and an initial value as argument, and returns a getter whose value changes when a new value is signaled by OSC:

```
v = osc.float("/volume", 1.)
s = amplify(v, s)
```

- `osc.on_float` allows registering a function which is called when a new value is received through OSC:

```
v = ref(1.)
osc.on_float("/volume", fun (x) -> v := x)
s = amplify({!v}, s)
```

- `osc.send_float` allows sending values through OSC:

```
s = amplify({!v}, s)
def send_volume()
  osc.send_float(host="1.2.3.4", port=7777, "/volume", !v)
end
thread.run(every=1., send_volume)
```

Of course, similar functions exist for other types (`osc.on_int`, `osc.send_bool`, etc.)

Comparing dry and wet. In order to test the pertinence of an effect, it is often useful to compare the sound without and with the effect. The `dry_wet` operator can help with this: it takes a float parameter, a source with the original sound (the *dry* source) and a source with the modified sound (the *wet* source). When the parameter varies between 0 and 1, the output varies between the dry and the wet source: with 0 only the dry source is played, with 1 only the wet source is played. For instance, if we want to test a compressor on a source `s`, we could have a setup such as

```
s2 = compress(threshold=-10., ratio=5., gain=4., s)
w  = interactive.float("wetness", min=0., max=1., 1.)
s  = dry_wet(w, s, s2)
output(s)
```

Here, `s` is the original source and `s2` is the source with the compressor effect applied. By varying the interactive variable `wet`, we can hear how compression affects the source.

6.6 Outputs

Now that we have the sound we were dreaming of for our radio, we are ready to export it to the world. We present here the various ways our stream can be distributed, as well as the various ways it can be encoded.

Soundcard output. The first basic output is the soundcard. As we have already seen many times, the `output` operator should select for us a decent soundcard output. You can also use various output operators, depending on the library you want to use to communicate with the soundcard `output.pulseaudio`, `output.alsa`, `output.portaudio` and `output.ao`. The first two are generally a good choice.

Dummy output. Liquidsoap also features an output called `output.dummy` which allows streaming to… nowhere! It can still be useful to animate source: without an output a source

does not produce a stream. As an illustration, suppose that we want to log the metadata (say, the title and artist) of a stream without listening to it. This could be performed as follows:

```
s = mksafe(input.http("http://my/favorite/radio"))
def log_song(m)
  file.write(append=true, data=json.stringify(m), "/tmp/songs")
end
s.on_track(log_song)
output.dummy(s)
```

The source `s` is here a distant stream fetched by `input.http`. Whenever we see a new track on `s`, we log the metadata in some file. The important part here is the last line: the use of `output.dummy` will make the source regularly produce a stream (as if it was connected to a soundcard output for instance) and we will thus be able to inspect tracks. If there is no output it is connected to, no one will ever ask `s` for data, and we would never see a track.

In fact, a script without any output will never do anything sensible with respect to streams, and for this reason Liquidsoap will simply refuse to start when there is no output, displaying the message

```
No output defined, nothing to do.
```

Icecast. Icecast is a server on which Liquidsoap can send a stream, which will take care of redistributing to the world. In order to use this method, you first need to setup such a server, which will not be detailed here: you can refer to the introductory material or the official documentation for this. We simply suppose here that we have setup a sever on the local machine (its address will be `localhost`) with the default password `hackme` (that you should really change if you do not want to run into problems).

Streaming our radio to the world is then as simple as this:

```
output.icecast(%mp3, host="localhost", port=8000,
               password="hackme", mount="my-radio.mp3", radio)
```

The `output.icecast` operator takes as first argument the encoding format: `%mp3` means that we want to encode our stream in MP3. The encoding formats are detailed in next section, for instance `%mp3.abr(bitrate=160)` would specify encoding in MP3 with average bitrate of 160 kbps, or `%fdkaac(bitrate=64)` would specify encoding in AAC format with bitrate of 64 kbps. Other arguments of `output.icecast` are: the host where Icecast is located, the port of the Icecast server (`8000` is the default port), the password to connect to the Icecast server, the mountpoint (this is the name of the radio for Icecast) and finally the source we want to encode (here, we suppose that our stream is named `radio`). We can then listen to the stream by connecting to the URL

```
http://localhost:8000/my-radio.mp3
```

The URL consists of the name of the Icecast server and its port, followed by the mountpoint. This allows streaming multiple radios in a same server, by giving them different mountpoint names. For instance, if we have a `rock` and a `techno` stream, we can encode both of them, and encode each of them both in MP3 and AAC with

```
out = output.icecast(host="localhost", port=8000, password="hackme")
out(%mp3,    mount="rock.mp3",   rock)
out(%fdkaac, mount="rock.aac",   rock)
out(%mp3,    mount="techno.mp3", techno)
out(%fdkaac, mount="techno.aac", techno)
```

Here, first define a function `out` which consists in `output.icecast` partially applied to the common parameters in order not to have to repeat them for each output, and then we define the various outputs. Note that it is absolutely not a problem that a given source is encoded multiple times (excepting perhaps that it requires some CPU resources).

Various arguments of `output.icecast` are available to provide more information about your radio including its `name`, `genre` and provide a `description` of it. The argument `dumpfile` can be useful to store the stream which is sent, in order to keep it for later inspection, although we would advise setting up a proper file output as described below.

Casting without ice. If you want to quickly test Icecast output without going through the hassle of setting up an Icecast server, you can use the `output.harbor` operator which will use Liquidsoap's internal webserver *harbor*. It will make Liquidsoap start a server which behaves like Icecast would, and it is as simple as this:

```
output.harbor(%mp3, mount="my-radio.mp3", radio)
```

As you can remark, for `output.harbor`, you only need to specify the encoding format, the mountpoint and the source to encode, and it will be available at

```
http://localhost:8000/my-radio.mp3
```

for you to listen. You can protect the stream by specifying a `user` and a `password` argument (both need to specified) which will then be required when trying to listen to the stream:

```
output.harbor(%mp3, mount="my-radio.mp3",
              user="bob", password="secret", radio)
```

Alternatively, you can also specify an authentication function in the `auth` argument: this function itself takes the user and password as arguments and returns whether the listener should be granted access to the stream. For instance, the following allows listeners whose password has odd length:

```
def auth(~address, login, password)
  log.important("Authentication from #{login} / #{password}")
  string.length(password) mod 2 == 1
end
output.harbor(%mp3, auth=auth, mount="my-radio.mp3", radio)
```

The arguments `on_connect` and `on_disconnect` are also useful to monitor connections from listeners.

HLS output. In the last few years, people have started moving away from Icecast and turn to HLS to distribute streams. Basically, a stream in HLS is a playlist of very short portions of the stream, called segments, whose duration is generally between 2 and 10 seconds. The playlist itself contains the last minute or so of the stream, split in segments, and is regularly updated. Compared to Icecast, this has the advantage of not requiring a constant connection from the user, and is thus robust to network changes or disconnections, and moreover, the segments are regular files and can thus be cached using the standard techniques for serving files over HTTP. Another useful feature of HLS is that multiple encodings of the same stream can be served simultaneously: typically, one would serve both a low and a high quality version of the stream, and the user can seamlessly switch between the two depending on the quality of its connection (e.g. when going from 3G to 5G on a phone).

The `output.file.hls` operator takes care of this. It takes as mandatory arguments the directory where all the files will be put (the playlist and the segments), a list of pairs of

stream names and encoding formats (remember that a stream can be encoded in multiple formats at once), and the source to encode. For instance, if we have a stream named `radio`, the script

```
output.file.hls("/tmp/hls",
  [("mp3-low", %mp3(bitrate=96)), ("mp3-hi", %mp3(bitrate=160))],
  radio)
```

will generate an HLS stream in the directory `/tmp/hls`, by encoding the source in two qualities (`mp3-low` which is MP3 encoded at the bitrate 96 kbps and `mp3-hi` which is MP3 encoded at 160 kbps). The directory `/tmp/hls` would then typically be served by an HTTP server. If you have a look at the contents of this directory, you will see that it contains

- a file `stream.m3u8`: this is the main playlist that your listeners should listen to (it links to streams in both qualities, between which the listener is able to choose),
- files `mp3-low.m3u8` and `mp3-hi.m3u8`: these are the playlists respectively corresponding to the stream in low and high quality,
- files `mp3-low_XX.mp3` and `mp3hi_XX.mp3`, where `XX` are numbers: these are the encoded segments, which are created and removed as the stream goes by.

Some useful arguments of the `output.file.hls` operator are the following.

- `encode_metadata`: whether to add metadata or not in the stream. This is disabled by default because some players assume that there will be one stream, and thus stop when they see metadata.
- `on_file_change`: this specifies a function which can be used to execute an action when a file is created or removed, which can typically to upload segments to a webserver when they are created and remove them when they are not in use anymore. This function takes an argument labeled `state` and the name of the file concerned. The `state` is a string which can be
  - `"opened"`: we have opened the file to start writing on it,
  - `"closed"`: we have finished writing to the file (and it could thus be uploaded to a server),
  - `"removed"`: we have removed the file (and it could thus be removed from the server).

  A simple example of such a function would be

  ```
  def on_file_change(~state, fname)
    if state == "closed" then
      print("The file #{fname} was created.")
    elsif state == "removed" then
      print("The file #{fname} was removed.")
    end
  end
  ```

  Here, we are only printing but, again, we would typically copy the files somewhere.
- `persist_at`: this specifies a file name which stores the state of the output (such as the currently created segments, in JSON format) and will be used to properly continue the HLS playlist in the case the script is stopped and restarted.
- `playlist`: the name of the main playlist, which is `"stream.m3u8"` by default.
- `segment_duration`: the duration of each segment, 10 seconds by default.

- `segment_name`: specify a function to generate the name of the segment files.
- `segments`: the number of segments per playlist, 10 by default.
- `segments_overhead`: the number of segments to keep, which are not anymore in the playlist, 5 by default. It might happen that some listeners take some time to download the files, and that they have an "old" version of the playlist, which will contain names for "old" segments. It is thus important to keep a few old segments in order to accommodate for such situations.
- `streams_info`: can be used to specify additional information about the streams such as the bandwith (in bits per second), the codecs (following RFC 6381), the extension for the files, and the dimensions in pixels for video streams.

A more involved example, inspired of `srt2hls`, is

```
aac_lofi  = %ffmpeg(format="mpegts",
                    %audio(codec="aac", channels=2, ar=44100))
aac_midfi = %ffmpeg(format="mpegts",
                    %audio(codec="aac", channels=2, ar=44100, b="96k"))
aac_hifi  = %ffmpeg(format="mpegts",
                    %audio(codec="aac", channels=2, ar=44100, b="192k"))
streams   = [("aac_lofi",  aac_lofi ),
             ("aac_midfi", aac_midfi),
             ("aac_hifi",  aac_hifi )]

def segment_name(~position, ~extname, stream_name) =
  timestamp = int_of_float(time())
  "#{stream_name}_#{timestamp}_#{position}.#{extname}"
end

output.file.hls(playlist          = "live.m3u8",
                segment_duration  = 2.,
                segments          = 10,
                segments_overhead = 5,
                segment_name      = segment_name,
                persist_at        = "state.config",
                "/tmp/hls", streams, radio)
```

It encodes the stream in AAC format, in three different qualities, with some custom parameters set up.

Encoders. Any encoder (see below) can be used for encoding HLS streams in Liquidsoap. However, the HLS specification enforces that the codecs used should be MP3 and AAC, so that you should restrict to those for maximum compatibility with players. Furthermore, in order to improve compatibility, it is recommended that sent data encapsulated in an `MPEG-TS` container: currently, the only encoder capable of doing this in Liquidsoap is `%ffmpeg`, as illustrated above.

Serving with Liquidsoap. It is possible to have Liquidsoap directly serve the files for the HLS stream with its internal web server with the operator `output.harbor.hls` (and `output.harbor.hls.ssl` for encrypting with SSL). The arguments of this operator are the same as those of `output.file.hls`, excepting `port` and `path` which respectively specify the port of the server, and the path where the stream is served. It is not recommended for

listener-facing setup, because we do not consider the internal web server harbor ready for heavy loads, but it can be useful to sync up with a caching system such as CloudFront. A simple setup would be

```
output.harbor.hls(port=8000, path="radio/", [("mp3", %mp3)], radio)
```

which would make the stream of the `radio` source available at the URL

```
http://localhost:8000/radio/stream.m3u8
```

File output. The next output we are going to see is the file output which, as you would expect, is performed by the operator `output.file`. It takes three arguments: the encoding format, the name of the file, and the source we want to encode in the file. For instance, we can encode a source `s` in the MP3 file `out.mp3` with

```
output.file(%mp3, "out.mp3", s)
```

The file name can contain special substrings which will automatically be replaced:

- some strings stand for the current time: `%Y` (year), `%m` (month), `%d` (day), `%H` (hour), `%M` (minute), `%S` (second), `%w` (weekday), `%z` (timezone),
- because of the above replacements, `%` is a special character: if you want to write `%` in a filename, you should write `%%` instead,
- some strings stand for the current metadata of the source: `$(title)` is the title, `$(artist)` is the artist, `$(album)` is the album, and so on.

For instance, when archiving a radio stream, it is useful to have the current time in the filename in order not to overwrite the file if the script is restarted:

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

The parameters `reopen_when` and `reopen_on_metadata` are particularly useful in association to this mechanism in order to generate multiple files. The parameter `reopen_when` allows to regenerate a new file (with an updated filename) when a predicate is true. For instance, we can generate an archive per hour with:

```
output.file(%mp3, "archive/%Y-%m-%d/%H-%M-%S.mp3", radio,
            reopen_when={0m})
```

Here, the predicate `{0m}` given for the `reopen_when` argument is true whenever the current minute is 0, i.e. at the beginning of every hour: we will thus change file at the beginning of every hour. Also note that the path of the archive file contains a directory which depends on the current date: Liquidsoap will take care of creating the required directories for us. Similarly, when the argument `reopen_on_metadata` is set to `true`, the file will be updated when some new track begins. For instance, we can create a new file for each track with the current date, artist and title with:

```
output.file(%mp3, "archive/%Y-%m-%d/$(artist) - $(title).mp3", radio,
            reopen_on_metadata=true)
```

The argument `on_close` can be used to specify a function which is called whenever we have finished writing to a file: this function takes the filename as argument. This is particularly useful to upload archive files to a backup server. For instance, in the script

```
def on_file(fname)
  log.important("Making a backup of #{fname}.")
  process.run("cp #{fname} /radio/backup/")
end
```

```
output.file(%mp3, "archive/%Y-%m-%d/%H-%M-%S.mp3", radio,
            reopen_when={0m}, on_close=on_file)
```

The function `on_file` is called each time an archive file is created. Here, we call a command to simply copy this file to the `/radio/backup` directory, but a more realistic application would for instance upload it on an FTP server or so.

Some other useful optional arguments of the `output.file` operator are

- `append`: when set to `true`, the file will not be overwritten if it exists, but new data will be added at the end instead,
- `fallible`: when set to `true` the operator will accept fallible sources, it will start recording the source when it is available and stop when this is not the case anymore,
- `on_start` and `on_stop` specify functions which are called whenever the source starts or stops.

The `on_stop` function is particularly useful when `fallible` is set to `true`: this allows calling a function when the source fails, see there for a concrete application.

YouTube. Another way to stream your radio to the world consists in using the usual video streaming platforms to deliver the contents. Let us illustrate this by sending our `radio` stream to YouTube (the setup for streaming to other platforms such as Twich or Facebook live should more or less be the same). This is done as follows:

```
radio = playlist("~/Music")
video = single("image.jpg")
radio = mux_video(video=video, radio)
ykey  = string.trim(file.contents("youtube-key"))
url   = "rtmp://a.rtmp.youtube.com/live2/#{ykey}"
enc   = %ffmpeg(format="flv",
          %video(codec="libx264", pixel_format="yuv420p",
                 b="300k", preset="superfast", r=25, g=50),
          %audio(codec="libmp3lame", samplerate=44100, b="160k"))
output.url(fallible=true, url=url, enc, radio)
```

The first thing we need to do here is to generate a video stream. Fancy ways to achieve this are detailed in this chapter. Here, we simply take an image `image.jpg`, generate a `video` stream from it and add it to the `radio` stream using the `mux_video` operator. Note that if you wanted to stream a video `video.mp4` instead of a static image, you could simply replace the second line by

```
video = single("video.mp4")
```

as expected. Now that our `radio` stream has both audio and video, we need to send it to YouTube. In order to do so, you first need to obtain the *stream key* from YouTube studio: this is the secret key which will allow us to send our stream. Since we do not like to put secrets in Liquidsoap scripts, we suppose that it is stored in the file `youtube-key` and read it in the variable `ykey`: the function `file.contents` returns the contents of the file and the `string.trim` function removes extraneous spaces or newlines that you might have put around the key. Finally, we specify in `enc` the way we want to encode video. Here, we use the FFmpeg encoder `%ffmpeg` which is further described in there, and encode the video in H.264 using the `libx264` encoder and the audio in MP3 using the `libmp3lame` encoder. These settings should get you started on most setups, however they require fine-tuning in order to improve quality, following explanations of there and there. In particular, the video bitrate given by `b="300k"` is very low in order to make sure that it will be working on any internet connection: if you have

a fast one we suggest that you increase this to something like `2000k` in order to have decent video quality. Finally, we store in `url` the address where we should stream (for YouTube, this is `rtmp://a.rtmp.youtube.com/live2/` followed by the key) and use the `output.url` operator to send our stream `radio` encoded with the encoder `enc` to the URL.

SRT. In order to send a stream on a local network, we recommend the use of the SRT protocol, using the `output.srt` operator, which has a low latency and can cope with network problems. This operator has two modes, which are specified by the `mode` argument.

- In `"caller"` mode, which is the default one, it initiates a connection to a remote server (specified by the parameters `host` and `port`). For instance,

  ```
  s = playlist("~/Music")
  output.srt(fallible=true, host="localhost", %wav, s)
  ```

  will connect to an SRT server on `localhost` on the default port `8000` (this client would typically be another instance of Liquidsoap with an `input.srt` input in `"listener"` mode) and stream the source `s` in WAV format.

- In `"listener"` mode, it waits for clients to connect to it to send them the stream. The `port` argument specifies the port it listens to. For instance, the script will send the stream of the source `s` encoded in MP3 when a client connects to it:

  ```
  s = playlist("~/Music")
  output.srt(fallible=true, mode="listener", %mp3, s)
  ```

  The stream can then be played by another Liquidsoap script with an `input.srt` in `"caller"` mode, or with external tools such as `ffplay`:

  ```
  ffplay srt://localhost:8000
  ```

6.7 Encoding formats

The encoding formats are specified by expressions of the form

```
%encoder
```

or

```
%encoder(parameters...)
```

if we need to specify some parameters. For instance, the encoding format for MP3, with default parameters, is

```
%mp3
```

and the format for MP3 at 192 kbps in joint stereo is

```
%mp3(bitrate=192, stereo_mode="joint_stereo")
```

This means that if we want to use this for a harbor output, we will write

```
output.harbor(%mp3(bitrate=192, stereo_mode="joint_stereo"),
              mount="my-radio.mp3", radio)
```

Liquidsoap has support for almost every common standard format. We explain here the main ones and refer to the online documentation for further details.

MP3. The MP3 format is perhaps the most widespread and well-supported compressed audio format. It provides reasonable quality and reasonable compression, so that it is often a good choice. There are three variants of the MP3 encoder depending on the way you want to handle bitrate (the number of bits of data per second the encoder will produce):

- `%mp3` or `%mp3.cbr`: constant bitrate encoding,
- `%mp3.vbr`: variable bitrate, quality-based, encoding,
- `%mp3.abr`: average bitrate based encoding,
- `%mp3.fxp` or `%shine`: constant bitrate fixed-point encoding.

The first one is predictable: it will always output the same amount of data. The second one is more adaptative: it will produce much data when the stream is "complex", and less when it is more "simple", which means that we get a stream of better quality, but whose bitrate is less predictable. The third one is a balance between the two: it will adapt to the complexity of the stream, but will always output the same bitrate on the average. You should rarely have to use the last one: it is a constant bitrate encoder, like `%mp3`, which does not use floating point computations (`fxp` stands for fixed-point), and is thus more suitable for devices without hardware support for floats, such as some low-end embedded devices.

The parameters common to all variants are

- `stereo` is either `false` or `true`: encode in mono or stereo (default is `stereo`),
- `stereo_mode` is either `"stereo"` or `"joint_stereo"` or `"default"`: encode left and right channels separately or conjointly (default is `"default"`),
- `samplerate` is an integer: number of samples per seconds in the encoded stream (default is `44100`),
- `internal_quality` is an integer between 0 and 9: controls the quality of the encoding, 0 being the highest quality and 9 being the worst (default is 2, the higher the quality the more CPU encoding takes),
- `id3v2` is either `false` or `true`: whether to add Id3v2 tags (i.e. metadata in our terminology) to the stream (default is `false`).

The parameters for `%mp3` are

- `bitrate`: the fixed bitrate in kilobits per second (kbps) of the encoded stream (common values are 128, 160 and 192 kbps, higher means better quality but also higher bandwidth).

The parameters for `%mp3.vbr` are

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

The parameters for `%mp3.abr` are

- `bitrate`: average bitrate (kbps),
- `min_bitrate`: minimun bitrate (kbps),
- `max_bitrate`: maximun bitrate (kbps),
- `hard_min`: minimal bitrate to enforce (kbps).

The parameters for `%mp3.fxp` are

- `channels`: the number of audio channels (typically 2 for stereo),
- `samplerate`: the desired samplerate (typically 44100),
- `bitrate`: the desired bitrate (in kbps, typically 128).

For instance, constant 128 kbps bitrate encoding is achieved with

```
%mp3(bitrate=128)
```

Variable bitrate with quality 7 and samplerate of 22050 Hz is

```
%mp3.vbr(quality=7, samplerate=22050)
```

Average bitrate with mean of 128 kbps, maximun bitrate 192 kbps and ID3v2 tags is

```
%mp3.abr(bitrate=128, max_bitrate=192, id3v2=true)
```

Fixed-point encoding in stereo at 44100 Hz at 128 kbps is

```
%mp3.fxp(channels=2, samplerate=44100, bitrate=128)
```

Wav. WAV is a non-compressed format: this means that you do not lose anything, but it takes quite some space to store audio. Not recommended for streaming, but rather for archiving. The parameters are

- `channels`: the number of channels (`1` and `2` can also respectively be specified with `mono` and `stereo`),
- `duration`: duration in seconds to set in the WAV header,
- `samplerate`: the number of samples per second,
- `samplesize`: the number of bits per sample (only the values 8, 16, 24 and 32 are supported for now, 16 being the reasonable default),
- `header`: whether a header should be issued or not (the value `false` means no header, and can be used for exchanging raw PCM data).

For instance,

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

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

Ogg. Liquidsoap has native support for Ogg which is a *container*: it is a file format which can contain multiple streams (typically, audio and/or video). The syntax for encoding in Ogg is `%ogg(...)`, where the `...` is a list of streams. The currently supported encoders for the streams themselves are Opus, Vorbis, Speex and FLAC for audio, and Theora for video. For instance, we can encode an Opus stream in an Ogg container with the encoder

```
%ogg(%opus)
```

For convenience, it is possible to simply write

```
%opus
```

instead of `%ogg(%opus)`, and similarly for other encoders. All Ogg encoders have a `bytes_per_page` parameter, which can be used to try to limit Ogg logical pages size, in bytes: this is the minimal amount of data which has to be read contiguously. For instance,

```
%opus(bytes_per_page=1024)
```

The usual value is between 4 kB and 8 kB.

Ogg/Opus. The Opus codec is an open-source codec intended as a modern replacement of both standard codecs (MP3, Vorbis) and highly compressed codecs (AAC, Speex). This is the one you should use by default for sound encapsulated in Ogg, unless you have specific needs.

It has the same or better quality than equivalent codecs and is free (both as in beer and as in speech). The only drawback is that it is slightly less supported on the user-end than MP3 and AAC, although it tends to be less and less the case.

The encoder is named `%opus` and its parameters are

- `samplerate`: samples per second (must be one of `8000`, `12000`, `16000`, `24000` or `48000`, default is 48000),
- `channels`: number of audio channels (must be `1` or `2`, default is 2, you can also write `mono` or `stereo` instead of `channels=1` or `channels=2` respectively),
- `vbr` specifies whether we want variable bitrate or not: it can either be `"none"` (for constant bitrate), `"constrained"` (for variable bitrate with constraints such as average target bitrate) or `"unconstrained"` (for unconstrained variable bitrate, which is the default),
- `bitrate`: encoding bitrate, in kilobits per second (between 5 and 512, can also be `"auto"` to leave the decision to the encoder which is the default),
- `signal`: can either be `"music"` or `"voice"` to specify the kind of input (this will influence the parameters of the encoder, by default the encoder regularly automatically determines the kind of input),
- `complexity`: the computational complexity between 0 (the fastest encoding, the lowest quality) and 10 (the slowest encoding and highest quality, which is the default),
- `frame_size`: encoding frame size, in milliseconds (must be one of `2.5`, `5.`, `10.`, `20.`, `40.` or `60.`), smaller frame sizes lower latency but degrade quality, default is 20,
- `application` specifies the target application: `"audio"` (the encoder determines it automatically, the default), `"voip"` (transmit voice over the internet) or `"restricted_lowdelay"` (lower the encoding delay down to 5 ms),
- `max_bandwidth` specifies the bandwidth of the sound to encode: can be `"narrow_band"` (for 3–4000 Hz), `"medium_band"` (for 3–6000 Hz), `"wide_band"` (for 3–8000 Hz), `"super_wide_band"` (for 3–12000 Hz) or `"full_band"` (for 3–20000 Hz), this is automatically detected by default,
- `dtx`: when set to `true`, the bitrate is reduced during silence or background noise,
- `phase_inversion`: when set to `false`, disables the use of phase inversion for intensity stereo, improving the quality of mono downmixes, but slightly reducing normal stereo quality.

More information about the parameters can be found in the Opus documentation. A typical encoder would be

```
%opus(samplerate=48000, bitrate=192, vbr="none", application="audio",
      channels=2, signal="music", max_bandwidth="full_band")
```

Ogg/Vorbis. Vorbis is an audio codec which was developed as an open-source replacement for MP3. It is now largely considered as superseded by Opus. There are three variants of the encoder:

- `%vorbis`: quality-based encoder,
- `%vorbis.abr`: encoder with variable bitrate,
- `%vorbis.cbr`: encoder with fixed constant bitrate.

The common parameters are

- `channels`: the number of audio channels (`mono` and `stereo` are also supported for 1 and 2 channels, the default is 2),
- `samplerate`: the number of samples per second (the default is 44100).

The parameters specific to `%vorbis` are

- `quality`: the quality of the stream between -0.1 (the lowest quality and smallest files) and 1 (the highest quality and largest files).

The parameters specific to `%vorbis.abr` are

- `bitrate`: the target average bitrate (in kilobits per second),
- `min_bitrate` and `max_bitrate`: the minimal and maximal bitrates.

The parameters specific to `%vorbis.cbr` are

- `bitrate`: the target bitrate (in kilobits per second).

For instance, a variable bitrate encoding can be achieved with

```
%vorbis(samplerate=44100, channels=2, quality=0.3)
```

an average bitrate encoding with

```
%vorbis.abr(samplerate=44100, channels=2, bitrate=128,
            min_bitrate=64, max_bitrate=192)
```

and a constant bitrate encoding with

```
%vorbis.cbr(samplerate=44100, channels=2, bitrate=128)
```

Ogg/Speex. The Speex codec is dedicated to encoding at low bitrates, targeting applications such as the transmission of voice over the internet, where having uninterrupted transmission of the stream, with low latency, is considered more important than having high-quality sound. It is now considered as superseded by the Opus codec.

The encoder is named `%speex` and its parameters are

- `samplerate`: the number of samples per second,
- `mono` / `stereo`: set the number of channels to 1 or 2,
- `abr`: encode with specified average bitrate,
- `quality`: use quality based encoding with specific value between 0 (the lowest quality) and 10 (the highest quality), default being 7,
- `vbr`: encode with variable bitrate,
- `mode` sets the bandwidth of the signal: either `"narrowband"` (8 kHz, default), `"wideband"` (16 kHz) or `"ultra-wideband"` (32 kHz),
- `complexity`: the computational complexity between 1 (the fastest encoding and lowest quality) and 10 (the slowest encoding and highest quality),
- `vad`: when set to `true` detects whether the audio being encoded is speech or silence/background noise,
- `dtx`: when set to `true` further reduce the bitrate during silence.

Ogg/FLAC. The last audio codec supported in the Ogg container is FLAC. Contrary to other codecs, it is a *lossless* one, which means that, after decoding, you get the exact same signal you encoded. However, the signal is still compressed in the sense that encoded sound takes less space than the raw data, as found for instance in the WAV format. By opposition, most other codecs are lossy: they deliberately forget about some parts of the signal in order to achieve higher compression rates.

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.

Note that contrarily to most other codecs, the two are not exactly the same.

The parameters are

- `channels`: the number of audio channels (2 by default, `mono` and `stereo` can also be used instead of `channels=1` and `channels=2`),
- `samplerate`: the number of samples per second (44100 by default),
- `bits_per_sample`: the number of bits per sample, must be one of 8, 16, 24 or 32, the default being 16,
- `compression`: the compression level between 0 (faster compression time, lower compression rates)) and 8 (slower compression time, higher compression rate), default being 5.

For instance,

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

A typical compression achieves around 75% of the original size for a rock song and 50% for a classical music song.

AAC. The AAC codec (AAC stands for *Advanced Audio Coding*) was designed to be a better replacement for MP3: it achieves better quality at the same bitrates and can decently encode the stream it low bitrates. Unlike Opus, its main competitor, patent license is required for distributing an AAC codec. However, it has better hardware support, especially on low-end devices.

The encoder is called `%fdkaac` and its parameters are

- `channels`: the number of audio channels (2 by default),
- `samplerate`: the number of samples per second,
- `bitrate`: encode at given constant bitrate,
- `vbr`: encode in variable bitrate with given quality between 1 (lowest bitrate and quality) to 5 (highest bitrate and quality),
- `aot`: specifies the *audio object type* (the kind of encoding for AAC data, which has influence on quality and delay): it can either be `"mpeg4_aac_lc"`, `"mpeg4_he_aac"`, `"mpeg4_he_aac_v2"` (the default), `"mpeg4_aac_ld"`, `"mpeg4_aac_eld"`, `"mpeg2_aac_lc"`, `"mpeg2_he_aac"` or `"mpeg2_he_aac_v2"`,
- `bandwidth`: encode with fixed given bandwidth (default is `"auto"`, which means that the encoder is free to determine the best one),
- `transmux`: sets the transport format, should be one of `"raw"`, `"adif"`, `"adts"` (the default), `"latm"`, `"latm_out_of_band"` or `"loas"`,
- `afterburner`: when set to `true` use *afterburner* which should increase quality, but also encoding time,
- `sbr_mode`: when set to `true`, use *spectral band replication*, which should enhance audio quality at low bitrates.

More information about the meaning of those parameters can be found in the hydrogenaudio knowledge base. For instance,

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

GStreamer. The `%gstreamer` encoder can be used to encode streams using the GStreamer multimedia framework, which handles many formats, and can provide effects and more on

the stream. It is quite useful, although its support in Liquidsoap should be considered as much less mature than the FFmpeg encoder, which fulfills similar purposes, and is presented next.

The parameters of the `%gstreamer` encoder are

- `channels`: the number of audio channels (2 by default),
- `log`: the log level of GStreamer between 0 (no message) and 9 (very very verbose), default is 5.

In GStreamer, *pipelines* describe sequences of GStreamer operators to be applied, separated by `!`, see also there. Those are specified by three further parameters of the encoder:

- `audio`: the audio pipeline (default is `"lamemp3enc"`, the LAME MP3 encoder),
- `video`: the video pipeline (default is `"x264enc"`, the x264 H.264 encoder),
- `muxer`: the muxer which takes care of encapsulating both audio and video streams (default is `"mpegtsmux"`, which performs MPEG-TS encapsulation).

If the `audio` pipeline is not empty then the number of audio channels specified by the `channels` parameter is expected, and if the `video` pipeline is not empty then one video channel is expected.

For instance, we can encode a source in MP3 with

```
%gstreamer(channels=2, audio="lamemp3enc", video="", muxer="")
```

The metadata of the encoded is passed to the pipeline element named `"metadata"` (the name can be changed with the `metadata` parameter of `%gstreamer`) using the "tag setter" API. We can thus encode our stream in MP3 with tags using

```
%gstreamer(audio="lamemp3enc", video="",
           muxer="id3v2mux name='metadata'")
```

or in Vorbis with tags using

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

Encoding a video in H.264 with MP3 audio encapsulated in MPEG transport stream is performed with

```
%gstreamer(audio="lamemp3enc", video="x264enc", muxer="mpegtsmux")
```

and in Ogg/Vorbis+Theora with

```
%gstreamer(audio="vorbisenc", video="theoraenc", muxer="oggmux")
```

The `audio`, `video` and `muxer` are combined internally to form one (big) GStreamer pipeline which will handle the whole encoding. For instance, with previous example, the generated pipeline is indicated with the following debug message:

```
[encoder.gstreamer:5] GStreamer encoder pipeline: appsrc name="audio_src"
↪  block=true caps="audio/x-raw, format=S16LE, layout=interleaved, channels=2,
↪  rate=44100" format=time max-bytes=40960 ! queue ! audioconvert ! audioresample
↪  ! vorbisenc ! muxer. appsrc name="video_src" block=true caps="video/x-raw,
↪  format=I420, width=1280, height=720, framerate=25/1, pixel-aspect-ratio=1/1"
↪  format=time blocksize=3686400 max-bytes=40960 ! 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 directly specify the whole pipeline. In this case, the parameter `has_video` is used to determine whether the stream has video or not (video is assumed by default). For instance, MP3 encoding can also be performed with

```
%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", has_video=false)
```

Beware that, when using the `%gstreamer` encoder, you should consider that the stream you are encoding is infinite (or could be). This means that not all containers (and muxers) will work. For instance, the AVI and MP4 containers need to write in their header some information that is only known with finite streams, such as the total time of the stream. These containers are usually not suitable for streaming, which is the main purpose of Liquidsoap.

FFmpeg. The `%ffmpeg` encoder is a "meta-encoder", just like the GStreamer one: it uses the versatile FFmpeg library in order to encode in various formats, including the ones presented above, but also many more. The general syntax is

```
%ffmpeg(format="<format>", ...)
```

where `<format>` is the container type and `...` is the list of streams we want to encode. All FFmpeg muxers should be supported as formats, the full list can also be obtained by running `ffmpeg -formats`. The special value `none` is also supported as format, in which case the encoder will try to find the best possible one.

Each stream can either be

- `%audio`: for encoding native audio (this is the one you generally want to use),
- `%audio.raw`: for encoding audio in FFmpeg's raw format,
- `%audio.copy`: for transmitting encoded audio (see below),
- `%video` / `%video.raw` / `%video.copy`: similar but for video (this will be developed in a later section).

The `%audio` and `%audio.raw` streams all take as parameters

- `codec`: the name of the codec to encode the stream (all FFmpeg codecs should be supported here, you can run the command `ffmpeg -codecs` to have a full list),
- `channels`: the number of audio channels,
- `samplerate`: the number of samples per second,

as well as parameters specific to the codec (any option supported by FFmpeg can be passed here). If an option is not recognized, it will raise an error during the instantiation of the encoder.

For instance, we can encode in AAC using the MPEG-TS muxer and the FDK AAC encoder, at 22050 Hz with

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

The profile `aac_he_v2` we use here stands for *high-efficiency* version 2 and is adapted for encoding at low bitrates. Here is a list of profiles you can use

- `aac_he_v2`: bitrates below 48 kbps,

- `aac_he`: bitrates between 48 kpbs and 128 kbps,
- `aac_low`: the default profile (*low complexity*), adapted for bitrates above 128 kbps.

We can encode in AAC in variable bitrate with

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

where `vbr` specifies the quality between 1 (the lowest quality and bitrate) and 5 (the highest quality and bitrate). We can encode MP3 at constant bitrate of 160 kbps with

```
%ffmpeg(format="mp3",
        %audio(codec="libmp3lame", b="160k"))
```

We can encode MP3 in variable bitrate quality 4 with

```
%ffmpeg(format="mp3",
        %audio(codec="libmp3lame", samplerate=44100, q=4))
```

where quality ranges from 0 (the highest quality, 245 kbps average bitrate) to 9 (the lowest quality, 65 kbps average bitrate). We can encode MP3 in variable bitrate with 160 kbps average bitrate with

```
%ffmpeg(format="mp3",
        %audio(codec="libmp3lame", b="160k", abr=1))
```

An encoding in Ogg/Opus with default parameters can be achieved with

```
%ffmpeg(format="ogg", %audio(codec="libopus", samplerate=48000))
```

Beware that the codec is `libopus` (not `opus` which is for standalone Opus) and that the default samplerate of 44100 is not supported in the Opus format (which is why we force the use of 48000 instead).

Some encoding formats, such as WAV, AVI or MP4, require rewinding their stream and write a header after the encoding of the current track is over. For historical reasons, such formats cannot be used with `output.file`. To remedy that, we have introduced the `output.url` operator. When using this operator, the encoder is fully in charge of the output file and can thus write headers after the encoding. The `%ffmpeg` encoder is one such encoder that can be used with this operator.

Encoded streams. By default, all the sources manipulate audio (or video) in Liquidsoap's internal format, which is raw data: for audio, it consists in sequences of samples (which are floats between -1 and 1). This is adapted to the manipulations of streams we want to make, such as applying effects or switching between sources. Once we have properly generated the stream, the outputs use encoders to convert this to compressed formats (such as MP3) which take less space. We detail here a unique feature of the FFmpeg encoder: the ability to directly manipulate encoded data (such as audio encoded in MP3 format) within Liquidsoap, thus avoiding useless decoding and re-encoding of streams in some situations. Note that most usual operations (even changing the volume for instance) are not available on encoded sources, but this is still quite useful in some situations, mainly in order to encode multiple times in the same format.

Remember that we can encode audio in MP3 format using FFmpeg with the encoder

```
%ffmpeg(format="mp3", %audio(codec="libmp3lame))
```

This says that we want to generate a file in the MP3 format, and that we should put in audio which is encoded in MP3 with the LAME library. Now, if we change this to

```
%ffmpeg(format="mp3", %audio.copy)
```

this says that we want to generate a file in the MP3 format, and that we should put in directly the stream we receive, which is supposed to be already encoded. The name `%audio.copy` thus means here that we are going to copy audio data, without trying to understand what is in there or manipulate it in any way.

Streaming without re-encoding. As a first example, consider the following very simple radio setup:

```
radio = playlist("~/Music")
output.icecast(fallible=true, mount="radio", %mp3, radio)
```

Our radio consists of a playlist, that we stream using Icecast in MP3. When executing the script, Liquidsoap will decode the files of the playlist in its internal format and then encode them in MP3 before sending them to Icecast. However, if our files are already in MP3 we are doing useless work here: we are decoding MP3 files and then encoding them in MP3 again to broadcast them. Firstly, this decoding-reencoding degrades the audio quality. And secondly, this is costly in terms of CPU cycles: if we have many streams or have more involved data such as video, we would rather avoid that. This can be done as explained above as follows:

```
radio = playlist("~/Music")
output.icecast(fallible=true, format="audio/mpeg", mount="radio",
               %ffmpeg(format="mp3", %audio.copy), radio)
```

Here, since we use the FFmpeg encoder with `%audio.copy`, Liquidsoap knows that we want `radio` to contain encoded data, and it will propagate this information to the `playlist` operator, which will thus not try to decode the audio files. Note that the `output.icecast` operator cannot determine the MIME type from the encoder anymore, we thus have to specify it by hand with `format="audio/mpeg"`.

Again, this is mostly useful for relaying encoded data, but we loose much of the Liquidsoap power, which does not know how to edit encoded data. For instance, if we insert the line

```
radio = amplify(0.8, radio)
```

in the middle, in order to change the volume of the `radio`, we will obtain the error

```
Error 5: this value has type
  source(audio=pcm(_),...)
but it should be a subtype of
  source(audio=ffmpeg.audio.copy(_),...)
```

which says that `amplify` is only able to manipulate audio data in internal format (`audio=pcm(_)`) whereas we have here encoded data (`audio=ffmpeg.audio.copy(_)`).

Now, suppose that, in addition to transmitting the MP3 files through Icecast, we also want to provide another version of the stream in Opus. In this, case we need to decode the stream provided by the source, which is encoded in MP3, before being able to convert it in Opus. This can be achieved with `ffmpeg.decode.audio`, which will decode any encoded stream into Liquidsoap internal format, after which we can handle it as usual:

```
radio = playlist("~/Music")
output.icecast(fallible=true, format="audio/mpeg", mount="radio",
```

```
              %ffmpeg(format="mp3", %audio.copy), radio)
radio = ffmpeg.decode.audio(radio)
# From there the radio source is decoded
output.icecast(fallible=true, format="audio/mpeg", mount="radio-opus",
              %opus, radio)
```

Incidentally, the functions `ffmpeg.decode.video` and `ffmpeg.decode.audio_video` are also provided to decode streams with video and both audio and video respectively.

Encode once, output multiple times. It sometimes happens that we want to have the same source, encoded in the same format, by multiple outputs. For instance, in the following script, we have a `radio` source that we want to encode in MP3 and output in Icecast, in HLS and in a file:

```
output.icecast(%mp3, mount="radio", radio)
output.file.hls("hls", [("mp3", %mp3)], radio)
output.file(%mp3, "radio.mp3", radio)
```

Because there are three outputs with `%mp3` format, Liquidsoap will encode the stream three times in the same format, which is useless and can be costly if you have many streams or, worse, video streams. We would thus like to encode in MP3 once, and send the result to Icecast, HLS and the file. This can be achieved by using the `ffmpeg.encode.audio` operator which will turn our source in Liquidsoap's internal format into one with encoded audio (using FFmpeg). This encoded audio can then be passed to the various outputs using the `%ffmpeg` encoder with `%audio.copy` to specify that we want to simply pass on the encoded audio. Concretely, the following script will encode the `radio` source once in MP3, and then broadcast it using Icecast, HLS and file outputs:

```
radio = ffmpeg.encode.audio(%ffmpeg(%audio(codec="libmp3lame")), radio)
# From there the radio source is in mp3 format
output.icecast(fallible=true, format="audio/mpeg",
              %ffmpeg(format="mp3", %audio.copy), mount="radio", radio)
output.file.hls(fallible=true, "hls",
               [("mp3", %ffmpeg(format="mp3", %audio.copy))], radio)
output.file(fallible=true, %ffmpeg(format="mp3", %audio.copy),
           "radio.mp3", radio)
```

For technical reasons, the output of `ffmpeg.encode.audio` is always fallible. We thus now have to pass the argument `fallible=true` to all outputs in order to have them accept this. As expected, Liquidsoap also provides the variants `ffmpeg.encode.video` and `ffmpeg.encode.audio_video` of the `ffmpeg.encode.audio` function in order to encode video sources or sources with both audio and video.

Note that the function `ffmpeg.decode.audio` can be thought of as an "inverse" of the function `ffmpeg.encode.audio`: this means that the script

```
s = ffmpeg.encode.audio(%ffmpeg(%audio(codec="libmp3lame")), s)
s = ffmpeg.decode.audio(s)
output(s)
```

will play the source `s`, after uselessly encoding it in MP3 and decoding it back to Liquidsoap's internal format for sources.

External encoders. Although the `%ffmpeg` encoder does almost everything one could dream of, it is sometimes desirable to use an external program in order to encode our audio streams.

In order to achieve this, the `%external` encoder can be used: with it, an external program will be executed and given the audio data on the standard input (as interleaved little-endian 16 bits integer samples), while we expect to read the encoded data on the standard output of the program.

The parameters of the `%external` encoder are

- `process`: the name of the program to execute,
- `channels`: the number of audio channels (2 by default),
- `samplerate`: the number of samples per seconds (44100 by default),
- `header`: whether a WAV header should be added to the audio data (`true` by default),
- `restart_on_metadata` or `restart_after_delay`: restart the encoding process on each new metadata or after some time (in seconds),
- `restart_on_crash`: whether to restart the encoding process if it crashed (this is useful when the external process fails to encode properly data after some time).

For instance, we can encode in MP3 format using the `lame` binary with the encoder

```
%external(process="lame - -")
```

Videos can also be encoded by external programs, by passing the flag `video=true` to `%external`: in this case, the data is given in AVI format on the standard input. For instance, a compressed Matroska file (with H.264 video and MP3 audio) can be produced using the `ffmpeg` binary with

```
%external(video=true,
  process="ffmpeg -f avi -i pipe:0 -f matroska \
           -c:v libx264 -c:a libmp3lame pipe:1")
```

6.8 Interacting with other programs

We now present the various ways offered by Liquidsoap in order to interact with other programs.

Sound from external sources. Let us first investigate the ways we provide in order to exchange audio data with other programs.

JACK. If the other program has support for it, the best way to exchange audio data is to use the *JACK library*, which is dedicated to this and provides very good performances and low latency. In order to use it, you first need to run a JACK server: this is most easily done by using applications such as *QjackCtl* which provides a convenient graphical interface. Once this is done, the applications using JACK export virtual ports which can be connected together.

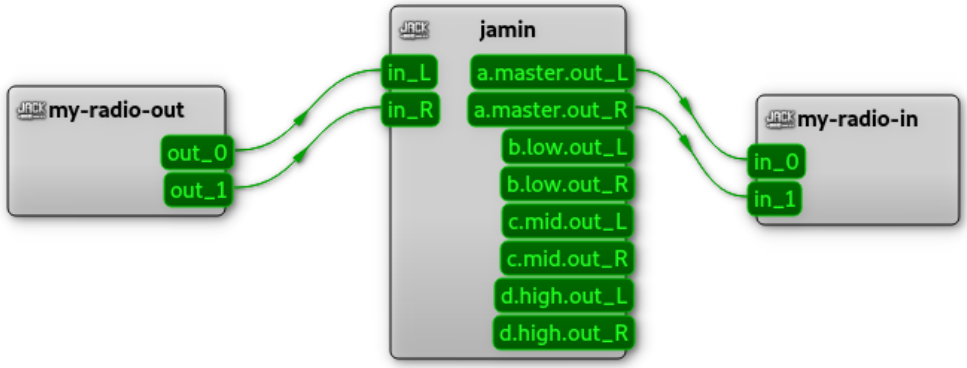
In Liquidsoap, you can use the operator `output.jack` to create a port to which the contents of a source will be streamed, and `input.jack` to create a port from which we should input a stream. When using this operators, it is a good idea to set the `id` parameter, which will be used as the name of the application owning the virtual ports.

For instance, in order to shape the sound of our radio, we might want to use *JAMin*, which is an audio mastering tool providing a multiband equalizer, a multiband compressor, a limiter, a spectrum analyzer and various other useful tools along with a graphical interface:

In order to do so, our script might look like this:

```
radio = mksafe(playlist("~/Music"))
output.jack(id="my-radio-out", radio)
radio2 = input.jack(id="my-radio-in")
output.icecast(mount="radio", %mp3, radio2)
```

We generate a `radio` source (here, this is a simple playlist), and send it to JACK using `output.jack`. Then we receive audio from JACK using `input.jack` and output it to Icecast. Finally, using an external tool such as QjackCtl we need to connect the output we created to JAMin, as well as the output of JAMin to our script:



External decoders/encoders. If JACK is not available, a more basic way of interacting with external tools is by the `pipe` operator, which runs the program specified in the `process` argument, writes audio to the standard input of the program and reads the audio from the standard output of the program. Both input and output are supposed to be encoded in WAV format. We have already seen this illustrated in there in order to use *Stereo Tool* to perform

mastering:

```
s = pipe(replay_delay=1.,
  process='/usr/bin/stereo_tool_cmd_64 - - -s myradio.sts -k "seckey"',
  s)
```

By default, the `pipe` operator launches a new process on each new track. However, this does not play well with Stereo Tool which needs to be run continuously: this is why we set `replay_delay` to 1, which means that we want to keep the same process across tracks and that metadata should be passed from the input to the output source with a delay of 1 second (which is approximately the delay introduced by the tool to process sound).

As another illustration, we can use the `ffmpeg` binary in order to amplify a source `s` with

```
s = pipe(process="ffmpeg -i - -filter:a volume=1.5 -f wav -", s)
```

Of course, in practice, using the builtin `amplify` operator is a much better idea if you simply want to change the volume.

Running external programs. We also have the possibility of executing external programs for performing various actions, by using the `process.run` function. For instance, suppose that we have a program `send-text-msg` to send a text message to the owner of the radio. The script

```
radio = blank.detect(
  {process.run("send-text-msg 'The radio is streaming blank!'")},
  radio)
```

uses the `blank.detect` operator, which calls a function when blank is detected on a source, and runs this program in order to alert that the radio has a problem.

The function `process.run` optionally takes an argument `timeout` which specifies the maximum number of seconds the program can take (if this is exceeded, the program is ended). It returns a record whose fields are as follows:

- `status`: a string describing how the program ended. It can either be
  - `"exited"`: the program exited normally (this is the value usually returned),
  - `"stopped"`: the program was stopped (it has received a STOP signal),
  - `"killed"`: the program was killed (it has received a KILL signal),
  - `"exception"`: program raised an exception.

  This field itself has two fields detailing the return value:
  - `code`: this is an integer containing the return code if the program exited, or the signal number if the program was stopped or killed,
  - `description`: a string containing details about the exception in the case the program raised one.
- `stdout`: what the program wrote on the standard output,
- `stderr`: what the program wrote on the standard error.

In the vast majority of cases, the program will return `"exit"` as status, in which case one should check whether the return code is 0 (the program ended normally) or not (the program ended with an error, and the status generally indicates the cause of an error leading to the end of the program, in a way which depends on the program). One thus typically checks for the end of the program in the following way:

```
p = process.run("my-prog")
if p.status == "exit" and p.status.code == 0 then
  print("The program exited normally.")
else
  print("An error happened while running the program: \
         #{p.status} #{p.status.code}.")
end
```

For convenience, the function `process.test` essentially implements this and returns whether the process exited correctly or not, which is convenient if you do not need to retrieve the standard output or error:

```
if process.test("my-prog") then
  print("The program exited normally.")
else
  print("An error happened.")
end
```

The field `stdout` of the returned value contains what the program printed on the standard output. For instance, in the script

```
p = process.run("ls -R ~/Music | wc -l")
if p.status == "exit" and p.status.code == 0 then
  n = int_of_string(string.trim(p.stdout))
  print("We have #{n} files in the library.")
end
```

we use `process.run` to execute the command

```
find ~/Music -type f | wc -l
```

which prints the number of files in the `~/Music` directory, we then read the standard output, use `string.trim` to remove the newline printed after the number, and then `int_of_string` to convert this value to an integer. Finally, we print the number of files. For convenience, the command `process.read` executes a command and returns its standard output, which is a bit shorter if we assume that the program will exit normally:

```
n = process.read("find ~/Music -type f | wc -l")
n = int_of_string(string.trim(n))
print("We have #{n} files in the library.")
```

Finally, the variant `process.read.lines` returns the list of all lines printed by the process on the standard output. For instance, the command

```
find ~/Music -type f
```

will list all files in the `~/Music` directory. We can use this to play all files in this directory as follows:

```
l = process.read.lines("find ~/Music -type f")
l = list.shuffle(l)
print("We are going to play #{list.length(l)} files.")
s = playlist.list(l)
output(s)
```

Here, `l` contains the list of all our files in the `~/Music` directory. We use `list.shuffle` to put it in a random order, print its length (obtained with `list.length`) and finally pass it to the `playlist.list` operator which will play all the files.

Security. As usual in the modern world, extra care should be taken when passing data from external users to applications, in order to mitigate the possibility of executing malicious code. This is typically the case if we use an external script in order to validate credentials for `input.harbor`. In such situations, one should always apply the function `string.quote` to the users' data, in order to remove the possibility that some parts of it are interpreted as bash commands. For instance,

```
def auth(login)
  process.test("./harbor-auth \
    #{string.quote(login.user)} #{string.quote(login.password)}")
end
s = input.harbor("live", auth=auth)
```

Following this practice should make your script pretty secure, but there is no way to be 100% sure that a corner case was not missed. In order to further improve security, Liquidsoap provides the possibility to *sandbox* processes, which means running them in a special environment which checks whether the directories the program reads from and writes to are allowed ones, whether it is allowed to use the network, and so on. In order to use this, one should set the `sandbox` configuration key as follows:

```
set("sandbox", true)
```

When this is done, every execution of a program by `process.run` (or derived functions) will be done using the sandboxing program `bwrap` (which can be changed with the configuration key `sandbox.binary`). The following configuration keys can then be set in order to change the permissions the run programs have by default:

- `sandbox.ro`: which directories the programs can read from (the root directory by default),
- `sandbox.rw`: which directories the programs can read from and write to (the home directory and the temporary directory by default),
- `sandbox.network`: whether programs have the right to use network (this it the case by default),
- `sandbox.shell`: whether programs have the right to run commands inside shell (this it the case by default).

The following arguments of the function `process.run`, can also be set for changing these values for a particular program instead of using the default settings as specified above:

- `rodirs`: which directories the program can read from (defaults to the value specified in `sandbox.ro` configuration key),
- `rwdirs`: which directories the program can read from and write to (defaults to the value specified in `sandbox.rw` configuration key),
- `network`: whether program has the right to use network (defaults to the value specified in `sandbox.network` configuration key).

For instance, suppose that the program `harbor-auth` we are using to authenticate harbor clients uses files in the directory `/users`. We can ensure that it only reads from there with

```
set("sandbox", true)
def auth(login)
  p = process.run(rodirs=["/users"], rwdirs=[], network=false,
        "./harbor-auth \
        #{string.quote(login.user)} #{string.quote(login.password)}")
  p.status == "exit" and p.status.code == 0
```

```
end
s = input.harbor("live", auth=auth)
```

In this way, even if a malicious user manages to use our authentication script to take control of our machine, he will not be able to access more than the list of users.

JSON. In order to exchange data with other programs (via `process.run`, files, and so on), the preferred way for formatting data is *JSON*, which is a standard way of representing structured data (consisting of records, arrays, etc.) and is supported by most modern languages. You can obtain the JSON representation of any Liquidsoap value with the function `json.stringify`, which takes a value as argument and returns its JSON representation. For instance, here is the way some Liquidsoap values are converted to JSON:

Liquidsoap	`()`	`true`	`"abc"`	`23`	`2.4`
JSON	`[]`	`true`	`"abc"`	`23`	`2.4`

Liquidsoap	`[2, 3, 4]`	`(12, 1.2)`
JSON	`[2, 3, 4]`	`[12, 1.2]`

Liquidsoap	`[("f", 1), ("b", 4)]`	`{x=1, y="a"}`
JSON	`{"f": 1, "b": 4}`	`{"x": 1, "y": "a"}`

The default output of `json.stringify` is designed to be pleasant to read for humans. If you want to have a small representation (without useless spaces and newlines), you can pass the argument `compact=true` to the function.

Conversely, JSON values can be converted to Liquidsoap using the `json.parse` function whose type is

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

It takes a string containing the JSON representation of a value and a `default` value, which is used both in order to determine the expected type for the value and to have a value to return by default in the case where the JSON data does not have the right type.

Parsing song requests. For instance, suppose that we have a script `next-song-json` which returns, in JSON format, the next song to be played along with cue in and out points and fade in and out durations. A typical output of the script would be of the form

```
{
  "file": "test.mp3",
  "cue_in": 1.1,
  "cue_out": 239.0,
  "fade_in": 2.5,
  "fade_out": 3.2
}
```

The following script uses `request.dynamic` which will call a function `next_song` in order to get the next song to be played. This function

- uses `process.read` to obtain the output of the external script `next-song-json`,
- uses `json.parse` with `default=[("","")]` in order to parse this JSON as a list of pairs of strings,
- extracts the parameters of the song from the returned list,
- returns a request from the song annotated with cue and fade parameters.

The cue and fade parameters are then applied by using the operators `cue_cut`, `fade.in` and `fade.out` because we annotated the parameters with the metadata expected by those operators.

```
def next_song()
  song     = process.read("./next-song-json")
  song     = json.parse(default=[("","")], song)
  file     = song["file"]
  cue_in   = song["cue_in"]
  cue_out  = song["cue_out"]
  fade_in  = song["fade_in"]
  fade_out = song["fade_out"]
  song = "annotate:liq_cue_in=#{cue_in},liq_cue_out=#{cue_out},\
          liq_fade_in=#{fade_in},liq_fade_out=#{fade_out}:#{file}"
  log.important("Next song is #{song}")
  request.create(song)
end
s = request.dynamic(next_song)
s = fade.out(fade.in(cue_cut(s)))
output(s)
```

Influence of the `default` *argument.* Note that the typing provided by `default` is important. Here, we parse the data with

```
song = process.read("./next-song-json")
song = json.parse(default=[("","")], song)
```

where the default value `[("","")]` is a list of pairs of strings and thus indicates that we want to parse it in this way, so that the returned value on the above input is

```
[("file", "test.mp3"),
 ("cue_in", "1.1"), ("cue_out","239."),
 ("fade_in","2.5"), ("fade_out","3.2")]
```

If we had parsed the JSON data as

```
song = process.read("./next-song-json")
song = json.parse(default=[("",0.)], song)
```

the default value `[("", 0.)]` would indicate that we want to parse it as a list of pairs of a string and a float, and we would have had the following Liquidsoap value as a result:

```
[("cue_in",1.1), ("cue_out",239.), ("fade_in",2.5), ("fade_out",3.2)]
```

Note that the fade parameters are given in floats instead of strings as in previous example, and that the field `file` is not present because there is no sensible way to parse it as a float.

Alternatively, we could also parse the JSON data as a record

```
song = process.read("./next-song-json")
default = {file="", cue_in=0., cue_out=0., fade_in=0., fade_out=0.}
```

```
song = json.parse(default=default, song)
```

which would give rise to the following record as result

```
{file="test.mp3", cue_in=1.1, cue_out=239., fade_in=2.5, fade_out=3.2}
```

and could therefore be used in order to provide the following alternative definition of the `next_song` function:

```
def next_song()
  song     = process.read("./next-song-json")
  default  = {file="", cue_in=0., cue_out=0., fade_in=0., fade_out=0.}
  song     = json.parse(default=default, song)
  print(string_of(fields=true, song))
  song = "annotate:\
          liq_cue_in=#{song.cue_in},liq_cue_out=#{song.cue_out},\
          liq_fade_in=#{song.fade_in},liq_fade_out=#{song.fade_out}:\
          #{song.file}"
  log.important("Next song is #{song}")
  request.create(song)
end
```

which is more structured and natural.

Watching files. A simple, although not very robust, way of communicating data with external programs is through files. For instance, suppose that we have a webserver on which users can make requests. When this is the case, our server writes the name of the file to play in a file `to-play`. In the following script, whenever the file `to-play` is modified, we push it in a queue so that it is played online, and we play the `default` source if there is no request:

```
queue   = request.queue()
radio   = fallback(track_sensitive=false, [queue, default])
def on_request()
  fname = string.trim(file.contents("to-play"))
  log.important("Playing #{fname}.")
  queue.push.uri(fname)
end
file.write(data=string_of(time()), "to-play")
file.watch("to-play", on_request)
output(radio)
```

In order to do so, we use the `file.watch` function, which registers a function to be called whenever a file changes: here, when the file `to-play` is modified, the function `on_request` is called, which reads the contents of the file and pushes a corresponding request on the queue. Of course, we could easily be combined this with the techniques of previous paragraph and store the file in JSON format to add additional information about the request.

If at some point you do not need to watch the file for changes anymore, the (unit) value returned by `file.watch` has a method `unwatch` which can be called to stop calling the function when the file is modified. For instance,

```
w = file.watch("to-play", on_request)
# ...
w.unwatch()
```

As another example, we could store the volume to be applied to our `radio` stream in a file named `volume` as follows:

```
volume = ref(1.)
radio  = amplify({!volume}, radio)
def update()
  v = string.trim(file.contents("volume"))
  volume := float_of_string(v)
  log.important("New volume is #{!volume}.")
end
file.watch("volume", update)
```

although this is more easily achieved using the `file.getter.float` function, as explained in an earlier section.

The telnet server. A common way of interacting between Liquidsoap and another program is through the telnet server, which can be used to by external programs to run commands in the scripts.

Configuration. In order to start the server, one should begin by setting the `server.telnet` configuration key to `true`:

```
set("server.telnet", true)
```

Other related configuration keys can be set:

- `server.telnet.bind_addr`: the IP from which the telnet server accepts commands (`"127.0.0.1"` by default, which means that only the local host can connect to the server, for security reasons),
- `server.telnet.port`: the port on which the server is listening (`1234` by default),
- `server.timeout`: timeout for read and write operations (30 seconds by default), if nothing happens for this duration the client is disconnected (setting this to a negative value disables timeout).

A running example. In order to illustrate the use of the telnet server we will be considering the following simple script, which implements a simple radio:

```
set("server.telnet", true)
q = request.queue(id="reqs")
p = playlist(id="main", "~/Music")
radio = fallback(track_sensitive=false, [q, p])
output(radio)
```

You can see that we have enabled telnet support and that our radio consists of a request queue named `reqs` with a fallback on a playlist named `main`.

Executing commands. We can connect to the server using the `telnet` program with

```
telnet localhost 1234
```

Here, `localhost` means that we want to connect on the local machine and `1234` is the default port for the server. Once this is done, we can begin typing commands and the server will answer to us. For instance, we can know the list of available commands by typing

```
help
```

to which the server will answer with

```
Available commands:
| exit
| help [<command>]
| list
| main.reload
| main.skip
| main.uri [<uri>]
| quit
| reqs.push <uri>
| reqs.queue
| reqs.skip
| request.alive
| request.all
| request.metadata <rid>
| request.on_air
| request.resolving
| request.trace <rid>
| uptime
| var.get <variable>
| var.list
| var.set <name> = <value>
| version

Type "help <command>" for more information.
END
```

The answer to a command can be arbitrary text, but always ends with a line containing only `END`, which is convenient when automating communications through telnet.

We have already seen that commands can also be sent with shell one-liners such as

```
echo reqs.skip | telnet localhost 1234
```

which will launch the `reqs.skip` command on the telnet server.

If you like web interfaces more than old shell programs, you can add

```
server.harbor()
```

in your script, and the telnet server will be available on your browser at the URL

```
http://localhost:8000/telnet
```

(the port and the URL can be configured by passing `port` and `uri` parameters to the function `server.harbor`). If you point your browser at this page, you should see a web emulation of the telnet server which looks like this:

Generic commands. Let us present the generic commands listed above, in the answer to the `help` command, i.e. the commands which will always be present in the telnet server of a script:

- `exit` ends the telnet communication,
- `help` prints the list of available commands, or prints detailed help about a command if called with a command name as argument:

  ```
  help version
  Usage: version
    Display liquidsoap version.
  END
  ```

  (the "usage" line explains how the command should be used, and which arguments are expected),
- `list` details the available operators, for instance, in our example, the answer would be

  ```
  reqs : request.dynamic.list
  main : request.dynamic.list
  switch_65380 : switch
  pulse_out(liquidsoap:) : output.pulseaudio
  ```

  indicating that we have two `request.dynamic.list` operators name `reqs` and `main`, a `switch` and an `output.pulseaudio` whose name have been automatically generated,
- `quit` is the same as `exit`,
- `uptime` shows for how long the script has been running,
- `version` displays the Liquidsoap version.

Some commands can be used to inspect the requests manipulated by Liquidsoap. Those are identified by their *request identifier*, or RID, which is a number uniquely identifying the request.

- `request.alive` lists all the requests which are in use, i.e. being played or waiting to be played,

- `request.all` lists all the requests used up to now,
- `request.metadata` can be used to list the metadata associated to a particular request,
- `request.on_air` lists all the requests which are being played,
- `request.resolving` lists all the requests which are being resolved, such as distant files being downloaded,
- `request.trace` shows the log associated to a particular request, which can be useful to know information about it, such as the reason why it failed to be resolved.

In a typical telnet session, we could ask for the alive and known requests:

```
request.alive
12 11
END
request.all
12 11 10 9
END
```

ask for the metadata of a particular request:

```
request.metadata 12
rid="12"
on_air="2021/05/20 17:04:06"
status="playing"
initial_uri="/path/to/file.mp3"
source="main"
temporary="false"
filename="/path/to/file.mp3"
title="My song"
artist="The artist"
kind="{audio=pcm(stereo),video=none,midi=none}"
END
```

trace a valid request:

```
request.trace 12
[2021/05/20 17:04:06] Pushed ["/path/to/file.mp3";...].
[2021/05/20 17:04:06] Currently on air.
END
```

push an invalid request and trace it:

```
reqs.push non-existent-file
13
END
request.trace 13
[2021/05/20 17:05:16] Pushed ["non-existent-file";...].
[2021/05/20 17:05:16] Nonexistent file or ill-formed URI!
[2021/05/20 17:05:16] Every possibility failed!
[2021/05/20 17:05:16] Request finished.
END
```

Some commands are specific to interactive variables and have been detailed in an earlier section:

- `var.get` provides the contents of an interactive variable,
- `var.list` lists all the defined interactive variables,

- `var.set` changes the value of an interactive variable.

Operators' commands. Above are presented the commands which are available in the telnet server of every script. But the operators used in a particular script also register additional commands. This is for instance the case for the `playlist` operator, which has registered the following three commands:

- `main.reload` reloads the playlist,
- `main.skip` skips the current song and goes to the next track,
- `main.uri` can be used to retrieve or change the location of the playlist.

Note that the commands are prefixed with `main`, which is the `id` of the playlist, so that we know which operator we are referring to (no prefix is added if no `id` is provided). The `request.queue` operator also has registered three commands

- `reqs.push` allows adding a new request in the queue, for instance

  ```
  reqs.push ~/Music/my file.mp3
  27
  END
  ```

  where the server returns the corresponding RID (`27` in our example),

- `reqs.queue` displays the list of requests in the queue,

- `reqs.skip` skips the current request in the queue.

Registering commands. You can register your own telnet commands with the `server.register` function. This function takes as argument the name of the command, and a function which will be called when the command is issued (the function receives as argument the argument on the command on telnet, and returns the message that will be printed after the command has been executed). Optional arguments labeled `usage` and `description` allow describing the way the command is intended to be used and what it does, and are used when displaying help.

For instance, suppose that we have three sources `rap`, `rock` and `techno`, and that we want to be able to switch between them whenever we want by typing the command "`select rap`", "`select rock`" or "`select techno`" on the telnet. This can be achieved as follows:

```
set("server.telnet", true)
selected = ref("techno")
def on_select(x)
  selected := x
  "Source #{x} selected."
end
server.register(usage="select <rap/rock/techno>",
                description="Switch between our music sources.",
                "select", on_select)
radio = switch(track_sensitive=false, [
  ({!selected == "rap"},    rap),
  ({!selected == "rock"},   rock),
  ({!selected == "techno"}, techno)
])
output(radio)
```

After enabling the telnet, we declare a reference `selected` to a string describing the currently selected source (it can be `"rap"`, `"rock"` or `"techno"`, and is initially the last one). We then define the callback function `on_select`, which changes the value of `selected` according to its argument. We then use `server.register` to have `on_select` be called when the `select` command is typed on the telnet. Finally, we define our radio with a `switch` which plays the `rap` source when the value of `selected` is `"rap"` and similarly for other sources. We can then change the played source to `rock` by typing the telnet command

```
select rock
```

to which the script will answer

```
Source rock selected.
END
```

As another example, the script

```
set("server.telnet", true)
radio = insert_metadata(radio)
def on_title(t)
  radio.insert_metadata([("title", t)])
  "Title set to #{t}."
end
server.register(usage="title <new title>",
                description="Set the title of the radio.",
                "title", on_title)
```

adds a command so that we can set the title of our `radio` stream by issuing a command of the form

```
title My new title
```

on the telnet server.

Interaction with other programs. The telnet server can be connected to using the usual means (TCP sockets), in almost every programming language. It is also possible to use commands such as `telnet` in order to send commands over the commandline. For instance:

```
echo select rock | telnet localhost 1234
```

When the web interface for the telnet server is enabled with `server.harbor()`, it is also possible to POST the command at the URL of the server (by default `http://localhost:8000/telnet`). You should have a look at the implementation of `server.harbor` in the standard library if you want to customize this (e.g. in order to support POST): it is based on `harbor.http.register`, which is described next.

Running commands from scripts. It is also possible to run a server command from within a Liquidsoap script itself by using `server.execute` function such as

```
server.execute("title My new title")
```

or

```
server.execute("title", "My new title")
```

if you want to separate the command from the argument. This is working even if the telnet interface for the server is not enabled.

Web-based interactions. A more and more common way of interacting with other programs and services nowadays is through the web, and Liquidsoap has support for this. Not only can we easily fetch webpages and distant files, but we also feature a builtin web server, called *harbor*, which can be handy in order to expose information on webpages or implement web services.

Making HTTP requests to other sites. We recall that Liquidsoap has integrated support distant files, in particular through the HTTP and HTTPS protocols. This means that you can load a playlist on some web server by writing something like

```
radio = playlist("http://www.some-server.com/playlist")
```

and the playlist can itself consist in a list of URL of files to be played.

It might also happen that you need to retrieve some distant file over HTTP and HTTPS. This can be achieved with the functions `http.get` which takes a URL as argument and returns the contents of the served page as a string. For instance, you can display the changelog for Liquidsoap with

```
#!./liquidsoap
c = http.get(
      "https://raw.github.com/savonet/liquidsoap/master/CHANGES.md")
print("Here are the latest changes in Liquidsoap:\n\n" ^ c)
```

The value returned by the function `http.get` also features the following fields, which can be used to obtain more information about the request:

- `headers` is the list of headers and their value,
- `protocol_version` is the version of the HTTP protocol we used (typically `"HTTP/1.1"`)
- `status_code` is a standard code for the HTTP protocol indicating the status of the answer (for instance, `200` means that everything went on fine and `404` means that the page was not found),
- `status_message` is a textual description of the status code.

When making requests, you should always check the status code in order to ensure that everything went on fine. A value above 400 means that an error occurred:

```
h = http.get("http://www.google.fr/xxx")
if h.status_code < 400 then
  print("Contents of the webpage: #{h}")
else
  print("An error occured: #{h.status_code} (#{h.status_message})")
end
```

Finally, the parameter `headers` of `http.get` can be used to pass extra headers to the request and the parameter `timeout` controls how long we can take at most in order to fetch the page (default is 10 seconds).

The HTTP protocol actually defines two main ways of retrieving webpages: POST, which is handled by the function `http.get` presented above, and POST, which is handled by the function `http.post`. The POST method is generally used for forms and takes an argument named `data`, which contains the data we want to pass as the contents of the form. The way this data is encoded is application-dependent and should be specified using the `Content-Type` header. For instance, suppose that we have a script `update_metadata.php` that we can call to update the metadata on our website. The script

```
radio = playlist("~/Music")
def handle_metadata(m)
  h = http.post(
        headers=[("Content-Type", "application/json; charset=UTF-8")],
        data=json.stringify(m),
        "http://our.website.com/update_metadata.php")
  if h.status_code >= 400 then
    log.important("Failed to update metadata.")
  end
end
radio.on_track(handle_metadata)
```

calls it whenever there is a new track, with the metadata of the track encoded in JSON as data.

Additional useful HTTP functions are `http.head` to only retrieve the headers of the corresponding POST request, `http.put` and `http.delete` which respectively perform PUT and DELETE HTTP requests in order to upload or delete a distant file.

Serving static webpages. Liquidsoap embeds a webserver which makes it possible for it to serve webpages. The most basic way of doing so is by making a directory available on the server using the `harbor.http.static` function. For instance, the line

```
harbor.http.static(port=8000, path="/music", browse=true, "~/Music")
```

will make all the files of the `~/Music` directory available at the path `/music` of the server, which will be made available on the port 8000 of the local host. This means that a file

```
~/Music/dir/file.mp3
```

will be available at the URL

```
http://localhost:8000/dir/file.mp3
```

The option `browse=true` makes it so that, for a directory, the list of files it contains is displayed. If the directory contains HTML pages, their contents will be displayed, so that this function can be handy to serve static pages. For instance, if we have a directory `www` containing a HTML file `test.html`, we can register it with

```
harbor.http.static(path="/pages", "www")
```

and browse it at

```
http://localhost:8000/pages/test.html
```

When serving files in this way, it is important that the server knows the kind of file it is serving. This is automatically detected by default, but it can also be specified manually with the `content_type` argument of `harbor.http.static`.

Serving dynamic webpages. The full power of the harbor server can be used through the `harbor.http.register` function, which allows serving HTTP requests with dynamically generated answers. It takes as arguments

- `port`: the port of the server (`8000` by default),
- `method`: the kind of request we want to handle (`"GET"`, `"POST"`, etc., default being `"GET"`),
- the URL we want to serve,
- the serving function.

This last function generates the answer for the request. Its type is

```
(protocol : string, headers : [string * string], data : string, string) -> {string}
```

which indicates that it receives as arguments

- `protocol`: the protocol for the request (e.g. `"HTTP/1.1"`),
- `headers`: the headers for the request,
- `data`: the input data (for POST requests),
- the URI we are serving,

and returns a string which is the HTTP answer. This answer has to follow a particular format specified by the HTTP protocol, and is usually generated by `http.response` which takes as argument the protocol (HTTP/1.1 by default), the status code (200 by default), the headers (none by default), the content type and the data of the answer, and properly formats it. For instance, in the script

```
def answer(~protocol, ~headers, ~data, uri)
  http.response(content_type="text/plain", data="It works!")
end
harbor.http.register("/test", answer)
```

we register the function `answer` on the URI `/test` which, when called, simply prints `It works!` as answer. You can test it by browsing the URL

```
http://localhost:8000/test
```

and you will see the message. We can also easily provide an answer formatted in HTML:

```
def answer(~protocol, ~headers, ~data, uri)
  http.response(content_type="text/html",
    data="<html><body><h1>It works!</h1></body></html>")
end
harbor.http.register("/test", answer)
```

Skipping tracks. What makes this mechanism incredibly powerful is the fact that the serving function is an arbitrary function, which can itself perform any sequence of operations in the script. For instance, we have already seen that we can implement a service to skip the current track on the source `s` with

```
def skipper(~protocol, ~headers, ~data, uri)
  s.skip()
  http.response(data="The current song was skipped!")
end
harbor.http.register(port=8000, "skip", skipper)
```

Whenever someone connects to the URL

```
http://localhost:8000/skip
```

the `skipper` function is called, and thus `s.skip()` is executed before returning a message.

Exposing metadata. The contents we serve might also depend on values in the script. For instance, the following script shows the metadata of our `radio` source encoded in JSON:

```
last_metadata = ref([])
radio.on_track(fun (m) -> last_metadata := m)
def show_metadata(~protocol, ~headers, ~data, uri)
```

```
  http.response(content_type="application/json; charset=UTF-8",
                data=json.stringify(!last_metadata))
end
harbor.http.register("/metadata", show_metadata)
```

We begin by declaring a reference `last_metadata` which contains the metadata for the last played track. Then, we register a callback so that whenever a new track occurs in `radio` we change the value of `last_metadata` according to its metadata. And finally, we register at the URL `/metadata` a function which returns a JSON encoding of the last metadata we have seen. As usual, the metadata can be retrieved by browsing at

```
http://localhost:8000/metadata
```

which will provide an answer of the following form:

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

This can be used with AJAX backends to fetch the current metadata of our radio.

Enqueuing tracks. We can also make use of the arguments of the serving function. For instance, we want that whenever we go to an URL of the form

```
http://localhost:8000/play?file=test.mp3&title=La%20bohème
```

we play the file `test.mp3` and specify that the title should be "La bohème". Here, the "real" part of the URL ends with `/play`, and the part after the question mark (`?`) should be considered as arguments, separated by ampersand (`&`) and passed in the form `name=value`. It should also be noted that an URL uses a particular, standardized, coding, where `%20` represents a space. This can be achieved as follows:

```
default = playlist("~/Music")
queue   = request.queue()
radio   = fallback(track_sensitive=false, [queue, default])
def play(~protocol, ~headers, ~data, uri)
  log.important("Serving uri: #{uri}")
  let (uri, args) = url.split(uri)
  fname = args["file"]
  title = args["title"]
  if file.exists(fname) then
    queue.push.uri("annotate:title=#{string.quote(title)}:#{fname}")
    http.response(data="Request pushed.")
  else
    http.response(code=404, data="Invalid file.")
  end
end
harbor.http.register("/play", play)
```

We begin by declaring that our radio consists of a requests queue with a fallback on a default playlist. We then register the function `play` on the URI `/play`. When we access the above URL, this function will receive

```
/play?file=test.mp3&title=La%20bohème
```

as `uri` argument. It uses `url.split` to split it into a pair consisting of the URI part (`/play`) and a list of arguments:

```
[("file", "test.mp3), ("title", "La bohème")]
```

This function also takes care of decoding the URL (`%20` was changed into a space in the title). Incidentally, if you need to decode an URL without splitting it, you can use the `url.decode` function (and conversely, the function `url.encode` encodes a string into an URL). Finally, the function pushes the corresponding request into the queue and answers that this has been performed.

Here, we validate the request by ensuring that the corresponding file exists. Generally, you should always validate data coming from users (even if you trust them, you never know), especially when passing data in requests without `string.quote` as it is the case here for `fname`.

Setting metadata. As a variant, we can easily make a script which, when an URL of the form

```
http://localhost:8000/play?artist=Charles%20Aznavour&title=La%20bohème
```

sets the current metadata of the `radio` source accordingly (the artist will be "Charles Aznavour" and the title "La bohème"):

```
radio = insert_metadata(radio)
def set_metadata(~protocol, ~headers, ~data, uri)
  log.important("Serving uri: #{uri}")
  let (uri, args) = url.split(uri)
  radio.insert_metadata(args)
  http.response(data="Done.")
end
harbor.http.register("/set_metadata", set_metadata)
```

This kind of mechanism can be handy when using inputs such as websockets, which do not natively support passing metadata.

Switching between sources. In an earlier section, we have seen how to switch between a `rock`, a `rap` and a `techno` source using telnet commands. Of course, this can also be achieved with a web service as follows:

```
selected = ref("techno")
def on_select(~protocol, ~headers, ~data, uri)
  let (uri, args) = url.split(uri)
  selected := args["source"]
  http.response(data="Source #{!selected} selected.")
end
harbor.http.register("/select", on_select)
radio = switch(track_sensitive=false, [
  ({!selected == "rap"},    rap),
  ({!selected == "rock"},   rock),
  ({!selected == "techno"}, techno)
])
```

Using this script, we can switch to the rap source by going to the URL

```
http://localhost:8000/select?source=rap
```

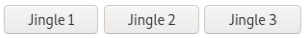
Launching jingles. Suppose that you want to be able to easily launch jingles during your show, with buttons which you could press at any time to launch a particular jingle. More precisely, we are interested here in playing a file among `jingle1.mp3`, `jingle2.mp3` and `jingle3.mp3`. In order to do this, we have prepared the following `jingle.html` file:

```
<html>
  <head>
    <script>function play(n) { fetch("?number="+n) }</script>
  </head>
  <body>
    <input type="button" value="Jingle 1" onclick="play(1)">
    <input type="button" value="Jingle 2" onclick="play(2)">
    <input type="button" value="Jingle 3" onclick="play(3)">
  </body>
</html>
```

Suppose that we serve this page at the URL

```
http://localhost:8000/jingles
```

When we go there, we see three buttons like this



Moreover, if we click on the button "Jingle 3", the page will fetch the URL

```
http://localhost:8000/jingles?number=3
```

and similarly for other buttons. Now, we can achieve what we want with the following script:

```
jingle_queue = request.queue()
radio        = add(normalize=false, [jingle_queue, radio])
def jingles(~protocol, ~headers, ~data, uri)
  log.important("Serving #{uri}")
  let (uri, args) = url.split(uri)
  n = list.assoc(default="", "number", args)
  if n != "" and string.is_int(n) then
    jingle_queue.push(request.create("jingle#{n}.mp3"))
    http.response(data="Playing jingle.")
  else
    http.response(content_type="text/html",
                  data=file.contents("jingles.html"))
  end
end
harbor.http.register("/jingles", jingles)
```

Here, we suppose that we already have a `radio` source. We begin by adding a queue `jingle_queue` on top of the radio. We then serve the URL `/jingles` with function `jingles`: if there is a `number` argument, we play the file `jingleN.mp3` where `N` is the number passed as argument, otherwise we simply display the page `jingles.html`.

Since we use a request queue, we cannot play two jingles at once: if we press multiple buttons at once, the jingles will be played sequentially. If instead of jingles you have some sound effects (for instance, laughter, clapping, etc.), you might want to play the files immediately. This can be achieved by using `request.player` instead of `request.queue` to play the jingles (and the method to play them is then `play` instead of `push`).

Limitations and configuration. When using Liquidsoap's internal HTTP server harbor, you should be warned that it 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.

Because of this, extra-care should be taken when exposing harbor. An external firewall should preferably be used, but the following configuration options can help:

- `harbor.bind_addrs`: list of IP addresses on which harbor should listen (default is `["0.0.0.0"]` which means any address),
- `harbor.max_connections`: maximum number of connections per port (default is 2 in order to mitigate the possibility of DDoS attacks),
- `harbor.ssl.certificate` and `harbor.ssl.private_key` should also be set if you want to use HTTPS connections.

6.9 Monitoring and testing

If you have read this (long) chapter up to there, you should now have all the tools to write the script for the radio you have always dreamed of. Now it is time to test this script to ensure that it performs as expected. We give here some functions that you can use to check that your script is running correctly.

Metrics. In order to ensure that your script is running alright at all times and perform forensic investigation in case of a problem, it is useful to have *metrics* about the script: these are data, often numeric data, which indicate relevant information about the stream production.

Useful indicators. The *power* of the stream can be obtained with the `rms` operator, which adds to a source an `rms` method which returns the current RMS. Here, RMS stands for *root mean square* and is a decent way of measuring the power of the sound. This value is a float between 0 (silent sound) and 1 (maximally loud sound). It can be converted to decibels, which is a more usual way of measuring power using the `dB_of_lin` function. For instance, the script

```
s = rms(s)
def print_rms()
  r = dB_of_lin(s.rms())
  log.important("RMS: #{r} dB")
end
thread.run(every=1., print_rms)
```

will print the power in decibels of the source `s` every second.

Another measurement for loudness of sound is LUFS (for *Loudness Unit Full Scale*). It is often more relevant than RMS because it takes in account the way human ears perceive the sound (which is not homogeneous depending on the frequency of the sound). It can be obtained quite in a similar way:

```
s = lufs(s)
thread.run(every=1., {print("LUFS #{s.lufs()}")})
```

The current BPM (number of *beats per minute*, otherwise known as tempo) of a musical stream can be computed in a similar way:

```
s = playlist("~/Music")
s = bpm(s)
thread.run(every=1., {print("BPM: #{s.bpm()}")})
output(s)
```

We can also detect whether the stream has sound or is streaming silence the using `blank.detect` operator:

```
silent = ref(false)
s = blank.detect(on_noise={silent := false}, {silent := true}, s)
thread.run(every=1., {log.important("Source is silent: #{!silent}")})
```

Other useful information for a particular source `s` can be obtained using the following methods:

- `s.is_up`: whether Liquidsoap has required the source to get ready for streaming,
- `s.is_ready`: whether the source has something to stream,
- `s.time`: how much time (in seconds) the source has streamed.

Exposing metrics. Once we have decided upon which metrics we want to expose, we need to make them available to external tools. For instance, suppose that we have a source `s` and that we want to export readyness, RMS and LUFS as indicators. We thus first define a function which returns the metrics of interest as a record:

```
s = rms(s)
rms = s.rms
s = lufs(s)
lufs = s.lufs
def metrics()
  {ready = s.is_ready(),
   rms   = dB_of_lin(rms()),
   lufs  = lufs()}
end
```

We can then export our metrics in a file, which can be performed with

```
def save_metrics()
  metrics = {rms = dB_of_lin(rms()), lufs = lufs()}
  data = json.stringify(metrics)
  file.write(data=data, "metrics.json")
end
thread.run(every=1., save_metrics)
```

Alternatively, metrics can be exposed using the webserver with

```
def metrics_page(~protocol, ~data, ~headers, uri)
  data = json.stringify(metrics())
  http.response(content_type="application/json", data=data)
end
harbor.http.register("/metrics", metrics_page)
```

Prometheus. If you need a more robust way of storing and exploring metrics, Liquidsoap has support for the Prometheus tool, which is dedicated to this task. Suppose that we have two sources named `radio1` and `radio2` for which we want to export the RMS. We first need to declare that we want to use Prometheus and declare the port we want to run the server on:

```
set("prometheus.server", true)
set("prometheus.server.port", 9090)
```

We are then going to declare a new kind of metric (here the RMS) using the function `prometheus.gauge` function:

```
rms_metric = prometheus.gauge(
  labels=["source"], help="RMS power", "liquidsoap_rms")
```

The type of `prometheus.gauge` is

```
(help : string, labels : [string], string) -> (label_values : [string]) -> (float)
↪  -> unit
```

this means that

- we first need to apply it to `help`, `labels` and the name of the gauge in order to create a new kind of gauge (here, the "RMS gauge"),
- we can apply the resulting function to `label_values` in order to create an instance of this gauge (we would typically do that once for every source of which we want to record the RMS),
- we finally obtain a function which takes a float as argument and can be used to set the value of the gauge.

In our case, we create two instances of the RMS gauge, one for each source:

```
set_radio1_rms_metric = rms_metric(label_values=["radio1"])
set_radio2_rms_metric = rms_metric(label_values=["radio2"])
```

Finally, we set the value of the gauges at regular intervals:

```
radio1 = rms(radio1)
radio2 = rms(radio2)
thread.run(every=1., {set_radio1_rms_metric(radio1.rms())})
thread.run(every=1., {set_radio1_rms_metric(radio2.rms())})
```

We also provide two variants of the function `prometheus.gauge`:

- `prometheus.counter` which increases a counter instead of setting the value of the gauge,
- `prometheus.summay` which records an observation.

Additional we provide the function `prometheus.latency` which can be used to monitor the internal latency of a given source.

On top of prometheus, Grafana offers a nice web-based interface. The reader interested in those technologies is advised to have a look at the `srt2hls` project which builds on Liquidsoap and those technologies.

Testing scripts. We provide here a few tips in order to help with the elaboration and the testing of scripts.

Logging. A first obvious remark is that you will not be able to understand the problems of your radio if you don't know what's going on, and a good way to obtain information is to read the logs, and write meaningful information in those. By default, the logs are printed on the standard output when you run a script. You can also have them written to a file with

```
set("log.file", false)
set("log.file.path", "/tmp/liquidsoap.log")
```

where the second line specifies the file those should be written to. A typical log entry looks like this:

```
2021/04/26 09:18:46 [request.dynamic_65:3] Prepared "test.mp3" (RID 0).
```

It states that on the given day and time, an operator `request.dynamic` (the suffix `_65` was added in case there are multiple operators, to distinguish between them) has issued a message at level `3` (important) and the message is "`Prepared "test.mp3" (RID 0).`", which means here that a request to the file `test.mp3` is about to be played and has RID 0.

We recall that there are various levels of importance for information:

1. a critical message (the program might crash after that),
2. a severe message (something that might affect the program in a deep way),
3. an important message,
4. an information, and
5. a debug message (which can generally be ignored).

By default, only messages with importance up to 3 are displayed, and this can be changed by setting the `log.level` configuration:

```
set("log.level", 5)
```

You can log at various levels using the functions `log.critical`, `log.severe`, `log.important`, `log.info` and `log.debug`. Those functions take an optional argument `label` which will be used as "operator name" in the log. For instance,

```
log.severe(label="testing", "This is my message to you.")
```

will add the following line in the logs:

```
2021/04/26 09:28:18 [testing:2] This is my message to you.
```

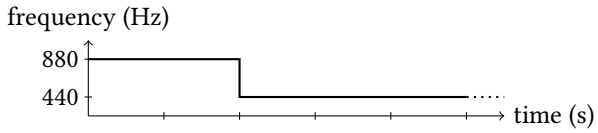
You should try to use priorities meaningfully: you will soon that at a high level such as 5 (debug) you get quite precise information, but the drawback is that you often actually get too much information and it is difficult to find the important one.

The telnet server (see above) can also be useful to obtain information such as the song currently playing on a particular source. In particular, `request.trace` can sometimes be used to understand why a particular request has failed to be played (be it because there was an error in the path or a failure of the network…).

Generating sound. Apart from reading the logs, the way you are generally going to test and debug your scripts is by using your ears. The simplest way to generate sound is by having a few music files at hand that you know well. Another very efficient way to generate easily recognizable sound is by generating sines, with different frequencies for different events. Those can be generated with the `sine` operator where the `duration` argument specifies the length of the track (infinite by default) and the unlabeled argument specifies the frequency. For instance, we can test the `fallback` operator as follows:

```
s1 = sine(duration=2., 880.)
s2 = sine(440.)
s  = fallback([s1, s2])
output(s)
```

We have two sources: `s1` is a high-pitched sine lasting for 2 seconds and `s2` is an infinite medium-pitched sine. We play the source `s` which is `s1` with a fallback on `s2`. If you listen to it, you will hear 2 seconds of high frequency sine and then medium frequency sine:



This can be particularly handy if you want to test faded transitions in fallback for instance. If you want to vary the kind of sound, the operators `square` and `saw` take similar parameters as `sine` and produce different sound waves.

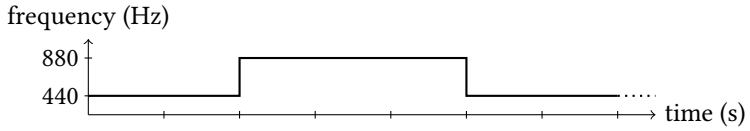
Sines can also be generated by special requests using the `synth` protocol, which are of the following form:

```
synth:shape=sine,frequency=440,duration=1
```

The parameters for the request are the shape of the generated sound wave (which can be `sine`, `saw`, `square` or `blank`, the default being `sine`), the frequency in Hz (440 by default) and the duration in seconds (10 by default). For instance, we can test request queues with the script

```
d = sine(440.)
q = request.queue()
s = fallback(track_sensitive=false, [q, d])
output(s)
thread.run(delay=2.,
           {q.push(request.create("synth:frequency=880,duration=3"))})
```

Our main source `s` consists of a request queue `q` with a fallback on a mid-pitched sine. After 2 seconds, we push on the request queue a high-pitch sine for 3 seconds. The frequency of what we hear will thus be



As a variant on sines, the `metronome` operator is sometimes useful: it generates sine beeps at fixed rate (one every second, or 60 BPM, by default). This can be used to measure time with your ears. For instance, in the above example, if you replace the definition of the default source `d` by

```
d = metronome()
```

you will be able to observe that the request is played slightly after the second second: this is because the request takes some time to be processed (we have to synthesize the sine!).

Generating events. The previous example should have made it clear that the function `thread.run` is quite useful to generate "events" such as pushing in a queue. Apart from the function to run `thread.run` takes two interesting arguments:

- `delay`: after how much time (in seconds) the function should be executed,
- `every`: how often (in seconds) the function should be called (by default, the function is only called once).

Typically, suppose that we want to test a function `handle_metadata` which logs the metadata of a source `s`. In order to test it, it can be boring to wait for the next track. We can use the telnet in order to skip the current track of the source by issuing a skip command. Even better, we can automate the skipping every 10 seconds with `thread.run` as follows:

```
s = playlist(id="s", "~/Music")
def handle_metadata(m)
  print("Metadata:\n#{json.stringify(m)}")
end
s.on_metadata(handle_metadata)
thread.run(every=10., {s.skip()})
output(s)
```

The `thread.run` function can be used to execute a function regularly according to the time of the computer. But, sometimes, it is more convenient to run the function according to the internal time of a particular source, which can be achieved with the `source.run` function: this function takes similar arguments as `thread.run`, but also a source which should be taken as time reference. For instance, suppose that you have a source `s2` which is in a fallback and that you want to skip it every 10 seconds when it is playing (and not skip when it is not playing). This is achieved with

```
s2 = source.run(s2, every=10., {s2.skip()})
s = fallback([s1, s2])
output(s)
```

In this example, the internal time of the source `s2` will not advance when it is not selected by `fallback`, and the source will thus not be skipped when this is the case.

Generating tracks. We have seen above that we can generate short tracks by regularly skipping a source. Since this is quite useful to perform tests (transitions, metadata handling, switching between sources, etc.), Liquidsoap provides various operators in order to do so.

- `skipper` takes a source and skips the track at regular intervals, specified by the argument `every`:

  ```
  s = skipper(every=3., s)
  ```

  (its implementation was essentially given above),

- `chop` takes a source and inserts track boundaries at regular intervals (specified by the argument `every`), with given metadata:

  ```
  s = chop(every=3.,
           metadata=[("artist", "Tester"), ("title", "Test")], s)
  ```

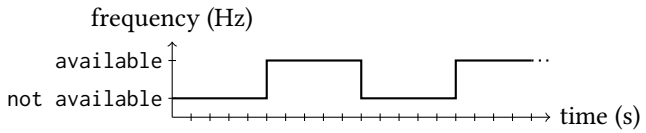
- `accelerate` plays a stream at faster speed by dropping or repeating frames, the speeding factor being given by the `ratio` argument, which specifies how many times we should speed up the stream:

  ```
  s = accelerate(ratio=10., s)
  ```

Availability of sources. We sometimes want to simulate sources which are not always available. For instance a live show which is only available when a client connects to some `input.habor`, or a microphone capture with `blank.strip(input.alsa())` which is only available when the microphone is switched on and someone is talking. Such a `live` source can be simulated using the `source.available` operator which makes a source available or not depending on a condition. For instance, in the script

```
live_on = ref(false)
live    = source.available(sine(), {!live_on})
pl      = playlist("~/Music")
radio   = fallback(track_sensitive=false, [live, pl])
output(radio)
thread.run(delay=5., every=5., {live_on := not !live_on})
```

we have a live source `live` with a fallback to a playlist. We use a thread to make the `live` source available only 5 seconds every 10 seconds:



Another way to achieve this is as follows:

```
live    = source.available(sine(), {time.up() mod 10. >= 5.})
pl      = playlist("~/Music")
radio   = fallback(track_sensitive=false, [live, pl])
output(radio)
```

We use the `time.up` function, which counts the time in seconds since the beginning of the execution of the script, with the `mod 10.` operator which forces the counting to go back to 0 every 10 seconds: thus `time.up() mod 10.` is greater than 5 for 5 seconds every 10 seconds, as desired.

The same tricks can be used to make a source silent for 5 seconds every 10 seconds, by amplifying with 1 or 0, in order to test `blank.strip` for instance:

```
def vol()
  if time.up() mod 10. <= 5. then 1.
  else 0. end
end
s = amplify(vol, playlist("~/Music"))
s = blank.strip(max_blank=2., s)
s = fallback(track_sensitive=false, [s, sine()])
output(s)
```

Simulating slow sources. In order to simulate sources which are slow to produce a stream (because of a high CPU load, because of a network lag, etc.), one can use the `sleeper` operator. It takes a `delay` operator which indicates how much time it should take to produce 1 second of audio. For instance, in the script

```
s = sleeper(delay=1.1, sine())
output(s)
```

we simulate a sine which takes 1.1 second to produce 1 second audio. Because the sound production is slower than realtime, you will soon hear glitches in the audio, as well as see log messages such as

```
2020/07/29 11:13:05 [clock.pulseaudio:2] We must catchup 0.86 seconds!
```

which indicate that the script is too slow to produce the stream.

A typical way to address those issues is to perform buffering, with the `buffer` operator, which will compute parts of the stream in advance in order not to be affected by small slowdowns. Liquidsoap also offers the `buffer.adptative` which will buffer and read the buffered data at low or high speed in order to accommodate for delays. This can be clearly heard in the following example:

```
s = sleeper(delay=1.1, random=.1, sine())
s = buffer.adaptative(s)
output(s)
```

You can hear that there are no audio glitches anymore, and that the frequency of the sine lowers progressively, because the buffered sound is progressively read more slowly in order to counter-balance the slow production of the sound.

Profiling. Liquidsoap features a builtin *profiler*, which records the time spent in all "pure Liquidsoap" functions (i.e. excluding encoding of audio and video). It can be enabled by calling `profiler.enable()` and then the profiling statistics can be retrieved at any time with `profiler.stats.string()`. This string is of the form

```
function     self              total             calls

string_of    0.110127687454    0.406255722046    46
+            0.108728647232    0.108728647232    52648
>=           0.0311367511749   0.0311367511749   2994
float_of_int 0.0238244533539   0.0497438907623   41
==           0.00842308998108  0.0166900157928   3
>            0.00465655326843  0.0109760761261   1
print        0.00128579139709  0.407711744308    45
```

and consists in a table, where each line corresponds to a function, and the columns indicate

- `function`: the function name,
- `self`: the time spent in the function, excluding the time spent in function calls,
- `total`: the time spent in the function, including the time spent in function calls,
- `calls`: the number of time the function was called.

For instance, in the script

```
rmsl = ref([])
s = rms(s)
s = source.run(s, every=0.01, {rmsl := list.add(s.rms(), !rmsl)})

def mean_rms()
  r = ref(0.)
  list.iter(fun (x) -> r := !r + x, !rmsl)
  !r / float_of_int(list.length(!rmsl))
end
```

```
s = source.run(s, every=1., {print("RMS: #{mean_rms()}")})

profiler.enable()
s = source.run(s, every=10., {print("#{profiler.stats.string()}")})
```

we want to compute the average RMS of a source `s`. We thus store the current RMS in a list `rmsl` every 0.01 second. Every second, we call a function `mean_rms` which computes the average of the list of `rmsl` and print the result. Every 10 seconds, we print the profiling statistics. You will see that, because the list `rmsl` grows over time, the script spends more and more time in the function `+` (in order to compute the average) and in the function `aux` (which is an internal function used to define `list.iter`).

6.10 Going further

A great way to learn more Liquidsoap tricks is to read the code of the standard library. For instance, we have already mentioned that even advanced functions are defined in there such as the `playlist` operator (in `playlist.liq`), interactive values (in `interactive.liq`) or the `normalize` operator (in `sound.liq`). In this section, we present some more advanced topics.

Operations on sources. If you have a look at the help of the function `sine`, you will notice that it has quite a number of methods. In fact, all the functions producing sources have those methods, some of which we have already seen, and we detail those here.

Some methods provide information about the source.

- `fallible`: indicates whether a source may fail or not. For instance,

  ```
  s = sine()
  print(s.fallible)
  ```

  will always print `false` because the sine source will never fail, but

  ```
  s = input.harbor("test")
  print(s.fallible)
  ```

  will print `true`, because a `input.harbor` source may fail (when nobody is connected to it).

- `id`: returns the identifier of the source.
- `is_up`: indicates whether the source has been prepared to start streaming.
- `is_ready`: indicates whether the source has something to stream (it should always be true for infallible sources).
- `remaining`: returns an estimation of remaining time in the current track.
- `time`: returns the source's time, i.e. for how long it has played.

Some methods allow registering functions called on some events.

- `on_metadata`: registers a function to be called on metadata (see there).
- `on_track`: registers a function to be called on new tracks (see there).
- `on_shutdown`: register a function to be called when the source shuts down.

Some methods allow performing actions on the source.

- `seek`: seek forward and returns the amount of time effectively seeked, see also earlier. The argument is given in seconds relative to current position, so that a negative value instructs seeking backward. Seeking is not available on every source (e.g. we cannot seek on an `input.http` source). The following script will loop in the first 10 seconds of the source `s`:

  ```
  set("decoder.priorities.mad",100)
  s = playlist("~/Music")
  thread.run(delay=10., every=10., {ignore(s.seek(-10.))})
  output(s)
  ```

  Namely, after 10 seconds of playing, we seek 10 seconds backwards. Here, we are giving high priority to the mad library to decode MP3 files, because the FFmpeg library, which is currently used by default, does not handle seeking very well.

- `skip`: skip to the next track.

- `shutdown`: deactivate a source.

Clocks. In order to avoid synchronization issues, Liquidsoap maintains *clocks*, which handle how the time is flowing for operators. Their behavior and usefulness is detailed in there, let us simply mention the two main causes of discrepancies between time flow between operators.

1. When performing an output (e.g. with `output.alsa`, `output.pulseaudio`, etc.) the speed at which the stream is generated is handled by the underlying library (ALSA, Pulseaudio, etc.). For this reason, all the different libraries have their own clock. In a script such as

   ```
   s = mksafe(playlist("~/Music"))
   output.alsa(s)
   output.pulseaudio(s)
   ```

   the source `s` would be played at two different rates, which would result in audio glitches. Liquidsoap detects this situation when the script is starting and issues the error

   ```
   A source cannot belong to two clocks (alsa[], pulseaudio[]).
   ```

   which that you are trying to animate the source `s` with both the `alsa` clock and the `pulseaudio` clock, which is forbidden.

2. Some operators need to change the time at which the source flows. This is for instance the case of the `stretch` operator which changes the speed at which a source is played or of the `crossfade` operator, which performs transitions between tracks, and thus needs to compute the next track in advance together with the end of a track. Such operators thus have their own clock. For this reason, the script

   ```
   s = mksafe(playlist("~/Music"))
   output.alsa(s)
   output.alsa(crossfade(s))
   ```

   will not be accepted either and will raise the error

   ```
   A source cannot belong to two clocks (alsa[], cross_65057[]).
   ```

   indicating that you are trying to animate the source `s` both at the `alsa` speed and at the crossfade speed (the number `65057` is there to give a unique identifier for each crossfade operator).

If the speed of a source is not controlled by a particular library or operator, it is attached to the `main` clock, which is the default one, and animated by the computer's CPU.

Buffers. In order to mediate between operators with two different clocks, one can use a buffer, which will compute the stream of a source in advance, and will thus be able to cope with small timeflow discrepancies. This can be achieved using the `buffer` operator which takes, in addition to the source, the following optional arguments:

- `buffer`: how much time to buffer in advance (1 second by default),
- `max`: how much time to buffer at most (10 seconds by default).

For instance, we can make the first script above work by adding a buffer as follows:

```
s = mksafe(playlist("~/Music"))
output.alsa(fallible=true, buffer(s))
output.pulseaudio(s)
```

Note that when executing it the ALSA output will be 1 second late compared to the Pulseaudio one: this is the price to pay to live in peace with clocks.

A typical buffer can handle time discrepancies between clocks between 1 and 10 seconds. It will not be able to handle more than this, if one source is going really too slow or too fast, because the buffer will be either empty or full. Alternatively, you can use `buffer.adaptative` which tries to slow down the source if it is too fast or speed it up if it is too slow. This means that the pitch of the source will also be changed, but this is generally not audible if the time discrepancy evolves slowly.

Deactivating clocks. Although we do not recommend it, in some situations it is possible to solve clock conflicts by deactivating the clock of a particular operator, often an input one. For instance, the script

```
s = input.alsa()
output.pulseaudio(s)
```

will not be accepted because the input and the output have different clocks, which are respectively `alsa` and `pulseaudio`. As indicated above, the standard way of dealing with this situation is by replacing the first line by

```
s = buffer(input.alsa())
```

However, there is another possibility: we can tell the `input.alsa` operator not to use its own clock, by passing the argument `clock_safe=false` to it.

```
s = input.alsa(clock_safe=false)
```

In this case, the output is the only operator with its own clock and will thus be responsible for the synchronization. This avoids using a buffer, and thus lowers latencies, which can be nice in a situation as above where we have a microphone, but this also means that we are likely to hear some glitches in the audio at some point, because the input might not be in perfect sync with the output.

Dealing with clocks. Apart from inserting buffers, you should almost never have to explicitly deal with clocks. The language however provides functions in order to manipulate them, in case this is needed. The function `clock` creates a new clock and assigns it to a source given in argument. It takes a parameter `sync` which indicates how the clocks synchronizes and can be either

- `"cpu"`: the clock follows the one of the computer,
- `"none"`: the clock goes as fast as possible (this is generally used for sources such as `input.alsa` which take care of synchronization on their own),
- `"auto"`: the clock follows the one of a source taking care of the synchronization if there is one or to the one of the CPU by default (this is mostly useful with `clock.assign_new`, see below).

Some other useful functions are

- `clock.assign_new`: creates a clock and assigns it to a list of sources (instead of one as in the case of `clock`),
- `clock.unify`: ensures that a list of sources have the same clock,
- `clock.status.seconds`: returns the current time for all allocated clocks.

Decoupling latencies. The first reason you might want to explicitly assign clocks is to precisely handle the various latencies that might occur in your setup, and make sure that delay induced by an operator do not affect other operators. Namely, two operators animated by two different clocks are entirely independent and can be thought of as being run "in parallel". For instance, suppose that you have a script consisting of a microphone source, which is saved in a file for backup purposes and streamed to Icecast for diffusion:

```
mic = input.alsa()
output.file(%mp3, "backup.mp3", mic)
output.icecast(%mp3, mount="radio", mic)
```

Here, all sources are animated by the same clock, which is the `alsa` one (because `input.alsa` is the only operator here which is able to take care of the synchronization of the sources). If for some reason, the Icecast output is slow (for instance, because of a network lag), it will slow down the `alsa` clock and thus all the operators will lag. This means that we might lose some of the microphone input, because we are not reading fast enough on it, and even the file output will have holes. In order to prevent from this happening, we can put the Icecast output in its own clock:

```
mic = input.alsa()
output.file(%mp3, "backup.mp3", mic)
output.icecast(%mp3, mount="radio", mksafe(clock(buffer(mic))))
```

In this way, the timing problems of Icecast will not affect the reading on the microphone and the backup file will contain the whole emission for later replace, even if the live transmission had problems. Note that we have to put a `buffer` operator between `mic` and the `clock` operator, which belong to different clocks, otherwise Liquidsoap will issue the usual error message

```
A source cannot belong to two clocks (alsa[], input.alsa_65308[]).
```

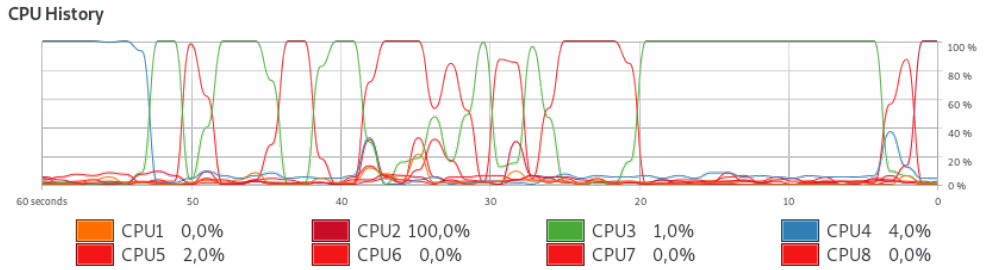
indicating that `mic` cannot belong both to its own new clock and the Icecast clock.

Encoding in parallel. From a technical point of view, each clock runs in its own thread and, because of, this two operators running in two different clocks can be run in parallel and exploit multiple cores. This is particularly interesting for encoding, which is the most CPU consuming part of the tasks: two outputs will different clocks will be able to encode simultaneously in two different cores of the CPU.

In order to illustrate this, consider the following script which performs two encodings of two different video files:

```
a = single("video.mkv")
b = single("video.mkv")
output.file(%theora, "/tmp/a.ogv", a)
output.file(%theora, "/tmp/b.ogv", b)
```

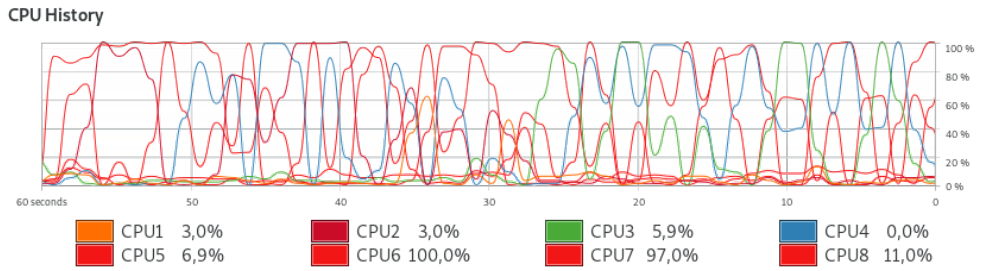
If we have a look at the CPU usage, we see that only one core is used at a given time:



(the kernel changes the core we use over time in order to better distribute the heat, but there is only one core used at a given time). Now, let us assign different clocks to the outputs, by changing the clock of the source of the second output, which will not be the default one anymore:

```
a = single("video.mkv")
b = single("video.mkv")
output.file(%theora, "/tmp/a.ogv", a)
output.file(%theora, "/tmp/b.ogv", clock(b))
```

We see that we now often use two cores simultaneously, which makes the encoding twice as fast:



Here, things are working well because the two encoders encode different sources (`a` and `b`). If they encode a common source, it can still be done by using a buffer

```
s = single("video.mkv")
output.file(%theora, "/tmp/a.ogv", s)
output.file(fallible=true, %theora, "/tmp/b.ogv", clock(buffer(s)))
```

with the risk that there is a glitch at some point because the speed of the clocks differ slightly, resulting in buffer under or overflow.

Offline processing. Liquidsoap has some support for writing standalone scripts, for instance to automate some processing on audio files. Those will typically begin with

```
#!/usr/bin/env -S liquidsoap -q
```

which means that the rest of the script should be executed by the default command `liquidsoap`, with the `-q` argument passed in order to avoid printing the usual log messages.

Retrieving commandline arguments. The arguments of the script can be obtained with the `argv` function which takes a number n and returns the n-th argument. This means that if your script is called `myscript.liq` and you run

```
./myscript.liq arg1 arg2
```

then `argv(1)` will return `"arg1"`, `argv(2)` will return `"arg2"` and `argv(3)` will return `""` (the empty string is returned by default when there is no such an argument). If you run your script with `liquidsoap` instead, the arguments in `argv` are those which are after the "`--`" argument. For instance, if we run

```
liquidsoap mylib.lib myscript.liq -- arg1 arg2
```

the arguments in `argv` are going to be `"arg1"` and `"arg2"`, as above.

Deactivating synchronization. Another useful trick to process files in scripts is to assign a clock to the output with `sync="none"` as parameter: this will make Liquidsoap assume that the operator is taking care of synchronization on its own, and thus produce the stream as fast as the CPU allows. This means processing a 5 minutes MP3 file will not take 5 minutes, but will be performed as fast as possible.

For instance, we have used this in the following script `convert2wav` which, as its name indicates, converts any audio file into WAV format:

```
#!/usr/bin/env -S liquidsoap -q

# Intput file name
infile = argv(1)
# Check that it exists
if infile == "" then
  print("Error: please provide a file as argument.")
  exit(1)
elsif not file.exists(infile) then
  print("Error: file #{infile} does not exist.")
  exit(1)
end
# Output file name
outfile = argv(default=path.remove_extension(infile)^".wav", 2)
print(newline=false, "Encoding #{infile} to #{outfile}... ")
# Play the file once
s = once(single(infile))
# We use a clock with disabled synchronization
s = clock(sync="none", s)
# Function called at the end of encoding
def stop()
  print("done!")
  shutdown()
end
# Encode the file
output.file(%wav, outfile, fallible=true, on_stop=stop, s)
```

It can be used to convert any file to a WAV file by running a command such as

```
convert2wav test.mp3
```

which will produce a file `test.wav`. As you can see, the script reads the file name as a commandline argument, plays it using `once(single(infile))` which is put in a clock without synchronization, and finally outputs it to a file, calling `shutdown` to stop the script when the whole file has been played and the source becomes unavailable.

Of course, we could easily adapt this script in order to apply audio effects to files (compression, volume normalization, etc), to merge a playlist into one file, and so on. For instance, the following `merge-playlist` script will merge a playlist of MP3 files into one MP3 file, without reencoding them:

```
#!/usr/bin/env -S liquidsoap
infile  = argv(1)
outfile = argv(default="playlist.mp3", 2)
s       = playlist(mode="normal", loop=false, infile)
s       = clock(sync="none", s)
output.file(fallible=true, on_stop=shutdown,
            %ffmpeg(format="mp3", %audio.copy), outfile, s)
```

It can be run with a commandline of the form

```
merge-playlist myplaylist output.mp3
```

Dynamic sources. Sources can be created dynamically in Liquidsoap: the number of source does not have to be fixed in advance. For instance, the script

```
pls = file.ls("playlists/")
def play(list)
  log.important("Playing #{list}.")
  s = mksafe(playlist("playlists/#{list}"))
  ignore(output.icecast(%mp3, mount="#{list}", s))
end
list.iter(play, pls)
```

will look at all the files in the `playlists/` directory, which are supposed to be playlists, and, for each such file (we iterate over the list of files with `list.iter`), will play it on a corresponding Icecast mountpoint. The point here is that the number of source is not fixed in advance as in most scripts: there will be as many `playlist` and `output.icecast` operators as there are playlists!

Creating sources during execution. Another point is that the creation of a source can be done at any point during the execution, not necessarily at startup. For instance, as a variant of the preceding script, we are going to register a telnet command `play p` which starts playing a given playlist `p` and `stop p` which stops playing a given playlist `p`, the playlist being located at `playlists/p`:

```
set("server.telnet", true)
sources = ref([])
def play(pl)
  s = mksafe(playlist("playlists/#{pl}"))
  s = output.icecast(%mp3, mount="#{pl}", s)
  sources := list.add((pl, s), !sources)
```

```
  "Playing #{pl}."
end
def stop(pl)
  s = list.assoc(pl, !sources)
  sources := list.assoc.remove(pl, !sources)
  s.shutdown()
  "Stopped #{pl}."
end
server.register("play", play)
server.register("stop", stop)
```

Here, we have two commands `play` and `stop` which are registered with `server.register`. We maintain a list `sources` which contains pairs consisting of the name of currently played playlists and the corresponding sources (this is an association list): the `play` command, in addition to playing the requested playlist, adds the pair `(pl, s)` to the list, where `pl` is the name of the playlist and `s` is the source we created to play it. The `stop` command finds the source to stop by looking for its name in the list with `list.assoc`, removes it from the list with `list.assoc.remove` and then stops the source by calling its `shutdown` method.

As a general rule, any dynamically created source which is not used anymore should be shut down using its `shutdown` method in order to avoid uselessly wasting resources.

Even more dynamic sources. We have seen that we can dynamically create sources, but this cannot be used to dynamically change the contents of sources which are already playing. This behavior can be achieved by using `source.dynamic`: this operator creates a source, which has a method `set` to change the source it is relaying, and can thus be used to create a source which changes overtime. It can be thought of as a suitable analogous of references for sources.

For instance, the `add` operator takes a fixed list of sources, but suppose that the list of sources we want to add varies over time. We can program this with `source.dynamic` where, each time the list changes, we use the `set` method to update the dynamic source and enforce that it consists in `add` applied to the updated list. For instance, the following script maintains a list `sine` sources, to which we add some new elements from time to time, with random parameters, by calling the function `new`:

```
d = source.dynamic()
sines = ref([])
output(d)

def new()
  # Shutdown and remove finished sinces
  sines := list.filter(remove=source.shutdown, source.is_ready, !sines)
  # Generate a new sine and add it to the list
  s = sine(amplitude=random.float(min=0., max=1.),
           duration=random.float(min=0., max=5.),
           random.float(min=200., max=1000.))
  sines := list.add((s:source), !sines)
  # Update the dynamic source
  d.set(add(normalize=false, !sines))
end
# Add a new sine from time to time
thread.run(every={random.float(min=0., max=2.)}, new)
```

Should you try it, you should hear music which sounds like random bells playing, sometimes interfering one with the other.

Of course, the above example is not something that you would usually have in a radio, but this is the same mechanism which is used to implement the `request.player` operator in the standard library. `source.dynamic` is also used in the standard library to provide alternative implementation of basic Liquidsoap operators, you should have a look at the file `native.liq` in order to learn more about those.

7 Video

Historically, Liquidsoap was dedicated to generating audio streams such as those found in radios, even though it was conceived from the beginning in order to be extensible with other kinds of data, such as video. When it started in 2004, there was absolutely no video support, then some work began to add that around 2009, but it was still not much used, partly because it was quite inefficient. Starting with the release of Liquidsoap 2.0 in 2021, the internal coding of video changed to RGB to YUV420, which is much more compact and used by most video libraries: Liquidsoap is now able to decently handle videos, as we will see in this chapter.

7.1 Generating videos

Playing a video. Most sources accepting audio files (`single`, `playlist`, etc.) also accept video files, so that generating a video stream is performed in the exact same way you would generate an audio stream, excepting that you start from video files instead of audio files. For instance, you can play a video file `test.mp4` with

```
s = single("test.mp4")
output.audio_video(s)
```

The operator `output.audio_video` plays both the audio and the video of the source `s`, and you can use `output.video` to play the video only. These operators chose a local output operator among the ones provided by Liquidsoap. There are currently two of them:

- `output.sdl` which uses the SDL library to display the video, and
- `output.graphics` which uses the library provided by OCaml in order to display graphical data.

The videos can even be directly be pulled from YouTube with the `youtube-dl` protocol, which requires that you have installed the `youtube-dl` program:

```
log.level.set(4)
server.harbor()
s = single(timeout=600.,
  "youtube-dl:https://www.youtube.com/watch?v=dQw4w9WgXcQ")
output.audio_video(s)
```

Since the whole video has to be downloaded beforehand, it can take quite some time, which is why we specify a "large" `timeout` parameter (10 minutes instead of the default 30 seconds).

As another example, if we have a playlist `video.playlist` of video files, it can be played with

```
s = mksafe(playlist("videos.playlist"))
output.graphics(s)
output.pulseaudio(s)
```

Generally, the video will be generated form a playlist using the `playlist` operator or from user's request using `request.queue` operator. Those were already presented in there, nothing changes for video.

The webcam. Under Linux, it is possible to use our webcam as a source with the `input.v4l2` operator which reads from the webcam:

```
s = input.v4l2()
output.video(s)
```

Parameters of the video. The format used by Liquidsoap for videos can be changed by setting the following configuration keys:

- `frame.video.width`: width of videos (in pixels),
- `frame.video.height`: height of videos (in pixels),
- `frame.video.framerate`: number of images per seconds.

The default format for images is 1280×720 pixels at 25 images per seconds which corresponds to the 720p (or *HD ready*) format. You can switch to 1080p (or *Full HD*) format with

```
set("frame.video.width",     1920)
set("frame.video.height",    1080)
set("frame.video.framerate",   25)
```

Remember that processing video data in realtime is very costly. Reducing the resolution to 854×480 (called 480p) or even 640×360 (called 360p) will degrade the quality of images, but can greatly improve the CPU consumption, in particular if your server is getting a bit old: a low resolution video is better than a laggy or jumpy one…

For convenience the functions `video.frame.width`, `video.frame.height` and `video.frame.rate` are also defined and return the corresponding configuration parameters.

Blank and colored frames. The operator `blank` can generate video (in addition to audio): it will generate an image which is *blank*, i.e. fully transparent. In order to generate a video of a given color, you can use the `video.fill` operator which fills the video of the source with the color specified in the `color` argument. For instance, the script

```
s = video.fill(color=0xff0000, blank())
output.video(s)
```

will play a red image. The color should be specified in hexadecimal, in the form `0xrrggbb` where `rr` specifies the red intensity, `gg` the green and `bb` the blue, each color ranges from `00` (color absent) to `ff` (color with maximum intensity) in hexadecimal.

Images. Images can be used as sources just as video files: they are accepted by operators such as `single`, `playlist`, etc. However, if you try the following simple script

```
s = single("test.png")
output.video(s)
```

Liquidsoap will complain that it cannot decode the file `test.png`. This is because, by default, Liquidsoap tries to decode the image with an audio track, and this is not possible for an image. We can however force the source to have no audio as follows, and you should then see the image:

```
s = (single("test.png"):source(audio=none))
output.video(s)
```

Here, `(x:source(audio=none))` means that we constrain `x` to be a source with no audio, this mechanism is explained in more details in there. In order for you to avoid thinking of those subtleties, the standard library provides the `image` operator which does this for you and conveniently creates a source from an image:

```
s = image("test.png")
output.video(s)
```

You are advised to use this operator when dealing with images.

Specifying the dimensions. Decoders also take in account the following metadata when decoding images:

- `x`, `y`: offset of the decoded image (in pixels),
- `width`, `height`: dimensions of the decoded image (in pixels),
- `duration`: how long the image is made available.

This means that the script

```
s = image("annotate:width=50,height=50:test.png")
output.video(s)
```

will show a small image of 50×50 pixels.

Cover art. Most recent audio formats (such as MP3 or Ogg) allow embedding the cover of the album into metadata. Liquidsoap has support for extracting this and provides the `video.cover` operator in order to extract the cover from an audio stream and generate a video stream from it. The script

```
a = playlist("~/Music")
s = mux_video(video=mksafe(video.cover(a)), a)
output.audio_video(s)
```

defines an audio source `a` from our music library, generates a video track `v` from its covers with `video.cover`, adds it to the sound track `a` (with `mux_video`, detailed below) and plays the result. It is important here that we use `mksafe` around `video.cover` in order to play black by default: the source will not be available when the track has no cover!

Playlists. If you want to rotate between images, you can use playlists containing images. However, remember that images have infinite duration by default, and therefore a `duration` metadata should be added for each image in order to specify how long it should last. The most simple way of performing this is to have entries of the form

```
annotate:duration=5:/path/toimage.jpg
```

Alternatively, if the playlist contains only the paths to the images, the `duration` metadata can be added by using the `prefix` argument of the playlist operator. For instance, the script

```
s = playlist(prefix="annotate:duration=2:", "image.playlist")
```

will display for 2 seconds the images of the playlist `image.playlist`.

Changing images. The `image` operator produces a source with a method `set` which can be used to change the displayed image: it takes as argument the new path to the image to stream. For instance, the following script shows a random image in the current directory every 2 seconds:

```
files = file.ls(".")
files = list.filter(string.match(pattern=".*\\.png|.*\\.jpg"), files)
s     = image(list.hd(files))
thread.run(every=2., {s.set(list.pick(files))})
```

In more details, the `file.ls(".")` function returns a list of files in the current directory. We then use `list.filter` to extract all the files which end with the `.png` or `.jpg` extension (the `string.match` function looks at whether the strings match the regular expression `.*\\.png|.*\\.jpg` which means: "anything followed by `.png` or anything followed by `.jpg`"). We define an `image` source `s` of which we change the image every 2 second using the `set` method, with `list.pick(files)` which picks a random element of the list `files`.

This mechanism can also be used to change the displayed image depending on some metadata. For instance, consider the script

```
a = playlist("playlist-with-images")
v = image()
a.on_track(fun(m) -> v.set(m["image"]))
s = mux_video(video=mksafe(v), a)
output.audio_video(s)
```

It creates a source `a` from a playlist `playlist-with-image` which contains audio songs with a metadata `image` indicating the image to display with the song. Typically, a line of this playlist would look like

```
annotate:image="myimage.png":mysong.mp3
```

(or the metadata `image` could also be hardcoded in the audio files). It also creates an `image` source `v`, whose image is set to the contents of the `image` metadata of each new track in `a`. Finally, we show the source `s` obtained by combining the audio source `a` and the video source `v`.

Adding videos. Our beloved `add` operator also works with videos. For instance, we can add a logo on top of our video source `s` by adding a scaled down version of our `logo.png` image:

```
s    = playlist("videos")
logo = image("annotate:x=10,y=10,width=50,height=50:test.png")
s    = add([s, logo])
output.audio_video(s)
```

When taking a list of sources with video as argument, the `add` operator draws the rightmost last: it is therefore important that the `logo` source is second so that it is drawn on top of the other one.

Because one often does this, Liquidsoap provides the function `video.add_image`, which allows adding an image on top of another source, and the previous script can be more concisely written as

```
s = playlist("videos")
s = video.add_image(x=10, y=10, width=50, height=50, file="test.png", s)
output.audio_video(s)
```

The function `video.add_image` moreover has the advantage of allowing getters for the parameters, so that we can program a moving logo as follows:

```
s = playlist("videos")
x = {int(10.+10.*cos(6.*time()))}
y = {int(10.+10.*sin(6.*time()))}
s = video.add_image(x=x, y=y, width=50, height=50, file="test.png", s)
output.audio_video(s)
```

Picture in picture. Instead of adding a small image on top of a big one, we can also add a small video on top of a big one. In order to reduce the size of a video, we can either use

- `video.scale`: which scales a video according to a given factor `scale`,
- `video.resize`: which resizes a video to a given size specified by its `width` and `height`.

Both functions also allow translating the video so that the upper-left corner is at a given position (`x`, `y`).

For instance, the following script adds a small webcam capture on top of the main video:

```
s = playlist("videos.playlist")
w = mux_audio(audio=blank(), input.v4l2())
w = video.scale(x=10, y=10, scale=0.1, w)
s = add([s, w])
output.audio_video(s)
```

Here, the main source `s` is a playlist of videos and `w` is the capture of the webcam. Since it does not have audio (only video), we add to it a blank audio track so that it has the same type as the source `s` and can be added with it. We scale down the webcam image with `video.scale` and finally add it on top of the main video with `add`.

Alpha channels. A defining feature of video in Liquidsoap is that *alpha channels* are supported for video: this means that images in videos can have more or less transparent regions, which allows to see the "video behind" whenever adding videos. The overall opacity of a video can be changed with the `video.opacity` operator, which takes a coefficient between 0 (transparent) and 1 (fully opaque) in addition to the source. For instance, with

```
s = add([s1, video.opacity(.75, s2)])
```

we are adding the source `s1` with the source `s2` made opaque at 75%: this means that we are going to see 75% of `s2`, and the remaining 25% are from `s1` behind.

Transparent regions are also supported from usual picture formats such as png. In particular, when you add a logo to a video stream, it does not have to be a square!

Combining audio and video sources. Given an audio source `a` and a video source `v`, one can combine them in order to make a source `s` with both audio and video with the `mux_audio` and `mux_video` operators, which respectively add audio and video to a source, by

```
s = mux_audio(audio=a, v)
```

or

```
s = mux_video(video=v, a)
```

For instance, we can generate a stream from a playlist of audio files and a playlist of image files with

```
a = playlist("~/Music")
v = playlist(prefix="annotate:duration=10:", "image.playlist")
s = mux_video(video=v, a)
```

The "opposite" of the muxing functions are the functions `drop_audio` and `drop_video`, which respectively remove the audio and video channels from a video track. For instance, if we have two sources `s1` and `s2` with both audio and video, we can create a source `s` with the audio from `s1` and the video from `s2` by

```
s = mux_audio(audio=drop_video(s1), drop_audio(s2))
```

(Cross)fading. In order to have nice endings for video, one can use the `video.fade.out` operator which will fade out to black (or actually rather to transparent) the video. The time it takes to perform this is controlled by the `duration` parameter (3 seconds by default), the kind of transition can be controlled by the `transition` parameter whose values can be

- `fade`: perform a fade to blank,
- `slide_left`, `slide_right`, `slide_up`, `slide_down`: make the video slide left, right, up or down,
- `grow`: makes the image get smaller and smaller,
- `disc`: have a black disc covering the image,
- `random`: randomly choose among the previous transitions.

Similarly, the operator `video.fade.in` add fade effects at the beginning of tracks:

```
s = video.fade.in(transition="disc", s)
```

Since the `add` and `cross` operators also work with video sources, this means that we can nicely crossfade the tracks of a video playlist as follows:

```
s = playlist("videos.playlist")
s = video.fade.out(duration=1., s)
s = video.fade.in (duration=1., s)
s = cross(duration=1.5, fun (a,b) -> add([a.source, b.source]), s)
output.audio_video(s)
```

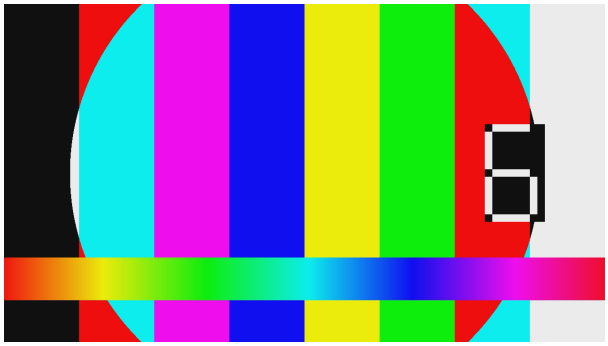
We apply fading at the beginning and the end of the videos, and then use the `cross` operator to add the end of each track with the beginning of the next one during 1.5 seconds. As a variant, slided transitions can be achieved with

```
s = playlist("videos.playlist")
s = video.fade.in (duration=1., transition="slide_right", s)
s = cross(duration=1., fun (a,b) -> add([a.source, b.source]), s)
output.audio_video(s)
```

Test sources. In order to generate test videos, the operator `video.testsrc` can be used. For instance,

```
s = video.testsrc()
```

will generate a video such as



The pattern displayed can be changed by passing the parameter `pattern` whose value can be `"testsrc"` (the default value), `"testsrc2"`, `"smptebars"` or `"rgbtestsrc"`.

Text. In order to add text on videos, we provide the `video.add_text` operator which, in addition to the text to print and the source on which it should add the text, takes the following optional arguments:

- `color`: color of the text, in the format `0xrrggbb` as explained above for `video.fill`,
- `font`: the path to the font file (usually in ttf format),
- `metadata`: metadata on which the text should be changed,
- `size`: the font size,
- `speed`: the speed at which it should scroll horizontally to have a "news flash" effect (in pixels per seconds, set to `0` to disable),
- `x` and `y`: the position of the text.

This function uses one of the various basic implementations we provide. You should actually try those various implementations in order to reach what you want: they have various quality and functionalities, and unfortunately we have not found the silver bullet yet. Those implementations are

- `video.add_text.native`: the native implementation. It always works and does not rely on any external library, but uses a hand-made, hard-coded, low-fi font.
- `video.add_text.sdl` / `video.add_text.gd` / `video.add_text.gstreamer` / `video.add_text.ffmpeg`: synthesize the text using SDL, GD, GStreamer and FFmpeg libraries.

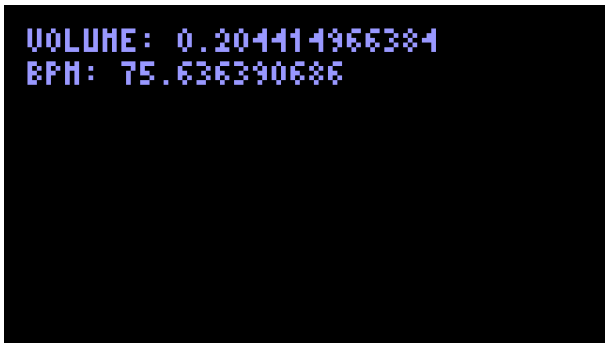
For instance,

```
s = video.add_text.sdl(size=30, "Hello world!", s)
```

The text is a getter which means that it can vary over times. For instance, the following prints the current volume and BPM of a song:

```
s = playlist("~/Music")
s = mux_video(video=blank(), s)
s = rms(s)
rms = s.rms
s = bpm(s)
bpm = s.bpm
s = video.add_text.native(color=0x9999ff, speed=0, x=50, y=50, size=50,
                          {"Volume: #{rms()}\nBPM: #{bpm()}"}, s)
output.audio_video(s)
```

and here is the output:



The position parameters are also getters, so that the position of the text can also be customized over time. For instance, the following will add a text moving along the diagonal at the speed of 10 pixels per second in each direction:

```
def x() = int(10. * time.up()) end
def y() = int(10. * time.up()) end
s = video.add_text(size=30, speed=0, x=x, y=y, "Hello world!", s)
```

7.2 Filters and effects

In order to change the appearance of your videos Liquidsoap offers video effects. These are not as well developed as for audio processing, but this is expected to improve in the future, and we support generic libraries which provide a large amount of effects.

Builtin filters. By default, Liquidsoap only offers some very basic builtin video filters such as

- `video.greyscale`: convert the video to black and white,
- `video.opacity`: change the opacity of the video,
- `video.fill`: fill the video with given color,
- `video.scale` / `video.resize`: change the size of the video.

Frei0r. Liquidsoap has native support for frei0r plugins, which are based on the frei0r API for video effects. When those are installed on your system, they are automatically detected and corresponding operators are added in the language. Those have names of the form `video.frei0r.*` where `*` is the name of the plugin. For instance, the following adds a "plasma effect" to the video:

```
s = video.frei0r.distort0r(s)
```

Each operator (there are currently 129) of course has specific parameters which allow modifying its effect, you are advised to have a look at their documentation, as usual.

FFmpeg filters. Another great provider of video (and audio) effects is FFmpeg: at the time of writing, we currently have access to 447 of those! Its filters are a bit more involved to use because FFmpeg expects that you create a *graph* of filters (by formally connecting multiple filters one to each other) before being able to use this graph for processing data, and because it operates on data in FFmpeg's internal format. Those filters can process both audio and

video data, we chose to present it here and not in previous chapter because it is more likely to be used for video processing.

The basic function we are going to use for creating filters is `ffmpeg.filter.create`. Its argument is a function `mkfilter` which takes as argument the graph of filters we are going to build, attaches filters to it, and returns the resulting stream. Usually this function

- uses the operators
  - `ffmpeg.filter.audio.input`
  - `ffmpeg.filter.video.input`

  to input from some stream,
- processes the stream using one or more `ffmpeg.filter.*` functions,
- outputs the result using one of the operators
  - `ffmpeg.filter.audio.output`
  - `ffmpeg.filter.video.output`
  - `ffmpeg.filter.audio_video.output`

In this way, we can define the following function `myfilter` which inputs the audio track and add a flanger effect to it, inputs the video track, flips its images horizontally and inverts the colors of the video, and finally outputs both audio and video:

```
def myfilter(s) =
  def mkfilter(graph) =
    a = ffmpeg.filter.audio.input(graph, s)
    a = ffmpeg.filter.flanger(graph, a)

    v = ffmpeg.filter.video.input(graph, s)
    v = ffmpeg.filter.hflip(graph, v)
    v = ffmpeg.filter.negate(graph, v)

    ffmpeg.filter.audio_video.output(graph, a, v)
  end
  ffmpeg.filter.create(mkfilter)
end
```

The function can then be used on a source `s` as follows:

```
s = single("lf.mp4")
s = myfilter(s)
output.file(fallible=true,
  %ffmpeg(format="matroska",
    %audio.raw(codec="libmp3lame"),
    %video.raw(codec="libx264")),
  "/tmp/out.mkv", s)
```

If you look at the type of the function `myfilter`, you will see that it is

```
(source(audio=ffmpeg.audio.raw('a), video=ffmpeg.video.raw('b), midi=none)) ->
↪  source(audio=ffmpeg.audio.raw('d), video=ffmpeg.video.raw('e), midi=none)
```

which means that it operates on streams where both audio and video are in FFmpeg's internal raw format (`ffmpeg.audio.raw` and `ffmpeg.video.raw`). In the above example this is working well because

- sources which decode audio from files such as `single` (or `playlist`) can generate streams in most formats, including FFmpeg's raw,
- the encoder we have chosen operates directly on streams in FFmpeg's raw format (because we use an `%ffmpeg` encoder with `%audio.raw` and `%video.raw` streams).

If you want to operate on a source `s` which is in the usual Liquidsoap's internal format, you can use

- `ffmpeg.raw.encode.audio_video` to convert from Liquidsoap's internal to FFmpeg's raw format,
- `ffmpeg.raw.decode.audio_video` to decode FFmpeg's raw format into Liquidsoap's internal format.

For instance, from the above `myfilter` function, we can define a function `myfilter'` which operates on usual streams as follows:

```
def myfilter'(s)
  s = ffmpeg.raw.encode.audio_video(%ffmpeg(%audio.raw, %video.raw), s)
  s = myfilter(s)
  ffmpeg.raw.decode.audio_video(s)
end
s = single("lf.mp4")
s = myfilter'(s)
output.audio_video(s)
```

by encoding before applying the filter and decoding afterward.

7.3 Encoders

The usual outputs described in there support streams with video, which includes

- `output.file`: for recording in a file,
- `output.icecast`: for streaming using Icecast,
- `output.hls`: for generating HLS playlists streams,
- `output.dummy`: for discarding a source.

We do not explain them here again: the only difference with audio is the choice of the encoder which indicates that we want to use sources with video.

FFmpeg. The encoder of choice for video is FFmpeg, that we have already seen in here. The general syntax is

```
%ffmpeg(format="...", %audio(...), %video(...))
```

where the omitted parameters specify the format of the container, the audio codec and the video codec.

Formats. The full list of supported formats can be obtained by running `ffmpeg -formats`. Popular formats for

- encoding in files:
  - `mp4` is the most widely supported, (its main drawback is that index tables are located at the end of the file, so that partially downloaded files cannot reliably be played, and the format is not suitable for streaming),

  - `matroska` corresponds to `.mkv` files, supports slightly more codecs than MP4 and it license-free, but is less widely supported,
  - `webm` is well supported by modern browsers (in combination with the VP9 codec),
  - `avi` is getting old and should be avoided,
- streaming:
  - `mpegts` is the standard container for streaming, this is the one you should use for HLS for instance,
  - `webm` is a modern container adapted to streaming with Icecast,
  - `flv` is used by some old streaming protocols such as RTMP, still widely in use to stream video to platforms such as YouTube.

Many other formats are also supported.

Codecs. The codec can be set by passing the `codec` argument to `%video`. The codecs all take `width` and `height` parameters, which allow setting the dimensions of the encoded video. Remember that smaller images have lower quality, but require smaller bitrates and encode faster. Common resolutions for 16:9 aspect ratio are

360p	480p	720p	1080p
640×360	854x480	1280×720	1920×1080

the "default reasonable value" being 720p nowadays. By default, the videos are encoded at the dimensions of internal frames in Liquidsoap, which can be set via `video.frame.width` and `video.frame.heigth`. If you only need to encode a video to "small" dimensions, it is a better idea to lower these values than specifying the codec parameters, in order to avoid computing large images which will be encoded to small ones.

You generally also want to set the bitrate by passing the `b` argument in bits per second (e.g. `b="2000k"`). Typical bitrates for streaming, depending on the resolution, at 25 frames per second, are

Resolution	Bitrate
640×360	700k
1280×720	2500k
1920×1080	4000k

Alternatively, many encoders allow specifying a "quality" parameter instead of a bitrate: in this case, it tries to reach a target quality instead of bitrate, by increasing the bitrate on complex scenes. This is not advised for videos intended for streaming since it can lead to unexpected bandwidth problems on those scenes.

Another useful parameter is the GOP (group of picture) which can be set by passing the argument `g` and controls how often keyframes are inserted (we insert one keyframe every `g` frames). A typical default value is 12, which allows easy seeking in videos, but for video streams this value can be increased in order to decrease the size of the video. The habit for streaming is to have a keyframe every 2 seconds or less, which means setting `g=50` at most for the default framerate of 25 images per second.

We now detail the two most popular codecs H.264 and VP9, but there are many other ones.

H.264. The most widely used codec for encoding video is `libx264` which encodes in H.264. This format has hardware support in many devices such as smartphones (for decoding). The most important parameter is `preset`, which controls how fast the encoder is, and whose possible values are

`ultrafast`, `superfast`, `veryfast`, `faster`, `fast`, `medium`, `slow`, `slower`, `veryslow`

with the obvious meaning. Of course, the faster the setting is the lower the quality of the video will be, so that you have to find a balance between CPU consumption and quality.

Instead of imposing a bitrate, one can also choose to encode in order to reach a target quality, which is measured in *CRF* (for Constant Rate Factor) and can be passed in the `crf` parameter. It is an integer ranging from 0 (the best quality) to 51 (the worse quality). In order to give you ideas,

- 0 is lossless,
- 17 is with nearly unnoticeable compression,
- 23 is the default value,
- 28 is the worse acceptable value.

Additional parameters can be passed in the `x264-params` parameter, e.g.

```
"x264-params"="scenecut=0:open_gop=0:min-keyint=150:keyint=150"
```

use this if you need very fine tuning for your encoding (you need to put quotes around the parameter name `x264-params` because it contains a dash).

A typical setting for encoding in a file for backup would be

```
%ffmpeg(format="mp4",
        %audio(codec="libmp3lame", q=4),
        %video(codec="libx264", preset="fast", crf=20))
```

and for streaming in HLS it would be

```
%ffmpeg(format="mpegts",
        %audio(codec="aac", b="96k"),
        %video(codec="libx264", preset="ultrafast", b="2500k"))
```

The successor of H.264 is called H.265 (how imaginative) or HEVC and is available through FFmpeg codec `libx265`. The parameters are roughly the same as those for `libx264` described above.

VP9 and AV1. VP9 is a recently developed codec, which is generally more efficient than H.264 and can achieve lower bitrates at comparable quality, and is royalty-free. It is supported by most modern browsers and is for instance the used by the YouTube streaming platform. It is generally encapsulated in the WebM container although it is supported by most modern containers.

The encoder in FFmpeg is called `libvpx-vp9`, some of its useful parameters are

- `quality` can be `good` (the decent default), `best` (takes much time) or `realtime` (which should be used in your scripts since we usually want fast encoding),
- `speed` goes from -8 (slow and high quality) to 8 (fast but low quality), for realtime encoding you typically want to set this to 5 or 6,
- `crf` controls quality-based encoding, as for H.264.

A typical WebM encoding would look like this:

```
%ffmpeg(format="webm",
        %audio(codec="libopus", samplerate=48000, b="128k"),
        %video(codec="libvpx-vp9", quality="realtime", speed=6,
               b="2500k"))
```

and if you are on budget with respect to CPU and bandwidth:

```
%ffmpeg(format="webm",
        %audio(codec="libopus", samplerate=48000, b="128k"),
        %video(codec="libvpx-vp9", width=854, height=480, g=75,
               quality="realtime", speed=7, b="500k"))
```

The successor of VP9 is AV1 and is under heavy development and diffusion. It can be used through the FFmpeg codec `libaom-av1` which essentially takes the same parameters as `libvpx-vp9`.

Ogg/Theora. We have support for the Theora video codec encapsulated in Ogg container, already presented in there. The encoder is named `%theora` whose main parameters are

- `bitrate`: bitrate of the video (for fixed bitrate encoding, in bits per second),
- `quality`: quality of the encoding (for quality-based encoding, between 0 and 63),
- `width` / `height`: dimensions of the image,
- `speed`: speed of the encoder,
- `keyframe_frequency`: how often keyframes should be inserted.

For instance, we can encode a video in Ogg with Opus for the audio and Theora for the video with

```
%ogg(%opus, %theora(bitrate=1000000))
```

AVI. Liquidsoap has native (without any external library) builtin support for generating AVI files with the `%avi` encoder. The resulting files contain raw data (no compression is performed on frames), which means that we need to compute almost nothing but also that it will not be compressed. This format should thus be favored for machines which are tight on CPU but not on hard disk, for backup purposes:

```
output.file(%avi, "/tmp/backup.avi", s)
```

You can expect the resulting files to be huge and you will typically want to re-encode the resulting files afterward.

If you want to generate AVI files with usual codecs, you should use the FFmpeg encoder presented above. For instance,

```
%ffmpeg(format="avi",
        %audio(codec="libmp3lame", b="128k"),
        %video(codec="libx264",    b="2500k"))
```

7.4 Specific inputs and outputs

Standard streaming methods. The two standard methods for streaming video are the same as those which have already been presented for audio in there: they are Icecast (with `output.icecast`) and HLS (with `output.hls`). The only difference is that the encoder should be one which has support for video.

Streaming platforms. Another very popular way of streaming video is by going through streaming platforms such as YouTube, Twitch or Facebook. All the three basically use the same method for streaming. You first need to obtain a secret key associated to your account on the website. Then you should send your video, using the RTMP protocol, to some standard URL followed by your secret key, using the `output.url` operator, and that's it. Because of limitations of the RTMP protocol, videos should be encoded using the FLV container with H.264 for video and MP3 or AAC for audio.

YouTube. The streaming key can be obtained from the YouTube streaming platform and the URL to stream to is

```
rtmp://a.rtmp.youtube.com/live2/<secret key>
```

If we suppose that we have stored our key in the file `youtube-key`, we can stream a video source `s` to YouTube by

```
key = string.trim(file.contents("youtube-key"))
url = "rtmp://a.rtmp.youtube.com/live2/#{key}"
enc = %ffmpeg(format="flv",
              %audio(codec="libmp3lame", samplerate=44100, q=5),
              %video(codec="libx264", width=854, height=480,
                     b="800k", g=50, preset="veryfast"))
output.url(fallible=true, url=url, enc, s)
```

These settings are for quite low quality encoding. You should try to increase them depending on how powerful your computer and internet connection are.

Twitch. The streaming key can be obtained from the Twitch dashboard and a list of *ingesting servers* is provided (the URL you should send your stream to is obtained by appending your key to one of those servers). For instance:

```
key = string.trim(file.contents("twitch-key"))
url = "rtmp://cdg.contribute.live-video.net/app/#{key}"
enc = %ffmpeg(format="flv",
              %audio(codec="libmp3lame", samplerate=44100, b="128k"),
              %video(codec="libx264", width=854, height=480,
                     b="800k", g=50, preset="veryfast"))
output.url(fallible=true, url=url, enc, s)
```

Facebook. The URL and streaming key can be obtained from the Facebook Live Producer. According to recommendations, your video resolution should not exceed 1280×720 at 30 frames per second, video should be encoded in H.264 at at most 4000 kbps and audio in AAC in 96 or 128 kbps. Keyframes should be sent at most every two second (the `g` parameter of the video codec should be at most twice the framerate). For instance,

```
s = single("video.mkv")
key = string.trim(file.contents("facebook-key"))
url = "rtmps://live-api-s.facebook.com:443/rtmp/#{key}"
enc = %ffmpeg(format="flv",
              %audio(codec="aac", samplerate=44100, b="96k"),
              %video(codec="libx264", width=854, height=480,
                     b="800k", g=50, preset="veryfast"))
output.url(fallible=true, url=url, enc, s)
thread.run(every=1., {print(clock.status.seconds())})
```

Saving frames. In case you need it, it is possible to save frames of the video with the `video.still_frame` operator: this operator adds to a source a method `save` which, when called with a filename as argument, saves the current image of the video stream to the file. Currently, only bitmap files are supported and the filename should have a `.bmp` extension. For instance, the following script will save a "screenshot" of the source `s` every 10 seconds:

```
s = single("video.mp4")
s = video.still_frame(s)
thread.run(every=10., {s.save("/tmp/shot#{time()}.bmp")})
output.audio_video(s)
```

8 A streaming language

After reading this chapter, you should have been convinced you that Liquidsoap is a pretty decent general-purpose scripting language. But what makes it unique is the features dedicated to audio and video streaming, which were put to use in previous chapters. We now present the general concepts behind the streaming features of the language, for those who want to understand in depth how the streaming parts of the language work. The main purpose of Liquidsoap is to manipulate functions which will generate streams and are called *sources* in Liquidsoap. The way those generate audio or video data is handled abstractly: you almost never get down to the point where you need to understand how or in what format this data is actually generated, you usually simply combine sources in order to get elaborate ones. It is however useful to have a general idea of how Liquidsoap works internally. Beware, this chapter is a bit more technical than previous ones.

8.1 Sources and content types

Each source has a number of channels of

- *audio* data: containing sound,
- *video* data: containing animated videos,
- *MIDI* data: containing notes to be played (typically, by a synthesizer).

The last kind of data is much less used in practice in Liquidsoap, so that we will mostly forget about it. Moreover, each of those channels can either contain

- *raw* data: this data is in an internal format (usually obtained by decoding compressed files), suitable for manipulation by operators within Liquidsoap, or
- *encoded* data: this is compressed data which Liquidsoap is not able to modify, such as audio data in MP3 format.

In practice, users manipulate sources handling raw data most of the time since most operations are not available on encoded data, even very basic ones such as changing the volume or performing transitions between tracks. Encoded data was introduced starting from version 2.0 of Liquidsoap and we have seen in an earlier section that it is however useful to avoid encoding a stream multiple times in the same format, e.g. when sending the same encoded stream to multiple Icecast instances, or both to Icecast and in HLS, etc.

The type of sources is of the form

```
source(audio=..., video=..., midi=...)
```

where the "..." indicate the *contents* that the source can generate, i.e. the number of channels, and their nature, for audio, video and MIDI data, that the source can generate: the contents for each of these three is sometimes called the *kind* of the source. For instance, the type of `sine` is

```
(?amplitude : {float}, ?duration : float, ?{float}) -> source(audio=internal('a),
↪  video=internal('b), midi=internal('c))
```

We see that it takes 3 optional arguments (the amplitude, the duration and the frequency) and returns a source as indicated by the type of the returned value: `source(...)`. The parameters of `source` indicate the nature and number of channels: here we see that audio is generated in some internal format (call it `'a`), video is generated in some internal data format (call it `'b`) and similarly for MIDI. The contents `internal` does not specify any number of channels, which means that any number of channels can be generated. Of course, for the `sine` operator, only the audio channels are going to be meaningful:

- if multiple audio channels are requested, they will all contain the same audio consisting of a sine waveform, with specified frequency and amplitude,
- if video channels are requested they are all going to be blank,
- if MIDI channels are requested, they are not going to contain any note.

As another example, consider the type of the operator `drop_audio` which removes audio from a source:

```
(source(audio='a, video='b, midi='c)) -> source(audio=none, video='b, midi='c)
```

We see that it takes a source as argument and returns another source. We also see that that is accepts any audio, video and MIDI contents for the input source, be they in internal format or not, calling them respectively `'a`, `'b` and `'c`. The returned source has `none` as audio contents, meaning that it will have no audio at all, and that the video content is the same as the content for the input (`'b`), and similarly for MIDI content (`'c`).

Internal contents. Contents of the form `internal('a)` only impose that the format is one supported internally. If we want to be more specific, we can specify the actual contents. For instance, the internal contents are currently:

- for raw audio: `pcm`,
- for raw video: `yuva420p`,
- for MIDI: `midi`.

The argument of `pcm` is the number of channels which can either be `none` (0 audio channel), `mono` (1 audio channel), `stereo` (2 audio channels) or `5.1` (6 channels for surround sound: front left, front right, front center, subwoofer, surround left and surround right, in this order). For instance, the operator `mean` takes an audio stream and returns a mono stream, obtained by taking the mean over all the channels. Its type is

```
(source(audio=pcm('a), video='b, midi='c)) -> source(audio=pcm(mono), video='b,
↪  midi='c)
```

We see that the audio contents of the input source is `pcm('a)` which means any number of channels of raw audio, and the corresponding type for audio in the output is `pcm(mono)`, which means mono raw audio, as expected. We can also see that the video and MIDI channels are preserved since their names (`'b` and `'c`) are the same in the input and the output.

Note that the contents `none` and `pcm(none)` are not exactly the same: for the first we know that there is no audio whereas for the second we now that there is no audio and that this is encoded in `pcm` format (if you have troubles grasping the subtlety don't worry, this is never useful in practice). For this reason `internal('a)` and `pcm('a)` express almost the same content but not exactly. Every content valid for the second, such as `pcm(stereo)`, is also valid for the first, but the content `none` is only accepted by the first (again, this subtle difference can be ignored in practice).

For now, the raw video format `yuva420p` does not take any argument. The only argument of `midi` is of the form `channels=n` where `n` is the number of MIDI channels of the stream. For instance, the operator `synth.all.sine` which generates sound for all MIDI channels using sine waves has type

```
(source(audio=pcm(mono), video='a, midi=midi(channels=16))) ->
↪  source(audio=pcm(mono), video='a, midi=midi(channels=16))
```

We see that it takes a stream with mono audio and 16 MIDI channels as argument and returns a stream of the same type.

Encoded contents. Liquidsoap has support for the wonderful FFmpeg library which allows for manipulating audio and video data in most common (and uncommon) video formats: it can be used to convert between different formats, apply effects, etc. This is implemented by having native support for

- the raw FFmpeg formats: `ffmpeg.audio.raw` and `ffmpeg.video.raw`,
- the encoded FFmpeg formats: `ffmpeg.audio.copy` and `ffmpeg.video.copy`.

Typically, the raw formats used in order to input from or output data to FFmpeg filters, whose use is detailed in there: as for Liquidsoap, FFmpeg can only process decoded raw data. The encoded formats are used to handled encoded data, such as sound in MP3, typically in order to encode the stream once in MP3 and output the result both in a file and to Icecast, this is detailed in there. Their name come from the fact that when using those, Liquidsoap simply copies and passes on data generated by FFmpeg without having a look into it.

Conversion from FFmpeg raw contents to internal Liquidsoap contents can be performed with the function `ffmpeg.raw.decode.audio`, which *decodes* FFmpeg contents into Liquidsoap contents. Its type is

```
(?buffer : float, ?max : float, source(audio=ffmpeg.audio.raw('a), video=none,
↪  midi=none)) -> source(audio=pcm('b), video=none, midi=none)
```

Ignoring the two optional arguments `buffer` and `max`, which control the buffering used by the function, we see that this function takes a source whose audio has `ffmpeg.audio.raw` contents and output a source whose audio has `pcm` contents. The functions `ffmpeg.raw.decode.video` and `ffmpeg.raw.decode.audio_video` work similarly with streams containing video and both audio and video respectively. The functions `ffmpeg.decode.audio`, `ffmpeg.decode.video` and `ffmpeg.decode.audio_video` have similar effect to decode FFmpeg encoded contents to Liquidsoap contents, for instance the type of the last one is

```
(?buffer : float, ?max : float, source(audio=ffmpeg.audio.copy('a),
↪  video=ffmpeg.video.copy('b), midi=none)) -> source(audio=pcm('c),
↪  video=yuva420p('d), midi=none)
```

Conversely, the functions `ffmpeg.raw.encode.audio`, `ffmpeg.raw.encode.video` and `ffmpeg.raw.encode.audio_video` can be used to encode Liquidsoap contents into FFm-

peg raw contents, and the functions `ffmpeg.encode.audio`, `ffmpeg.encode.video` and `ffmpeg.encode.audio_video` can encode into FFmpeg encoded contents.

The parameters for the FFmpeg contents are as follows (those should be compared with the description of the raw contents used in Liquidsoap, described in there):

- `ffmpeg.audio.raw`
  - `channel_layout`: number of channels and their ordering (it can be `mono`, `stereo` or `5.1` as for Liquidsoap contents, but many more are supported such as `7.1` or `hexagonal`, the full list can be obtained by running the command `ffmpeg -layouts`)
  - `sample_format`: encoding of each sample (`dbl` is double precision float, which is the same as used in Liquidsoap, but many more are supported such as `s16` and `s32` for signed 16- and 32-bits integers, see `ffmpeg -sample_fmts` for the full list),
  - `sample_rate`: number of samples per second (typically, 44100),
- `ffmpeg.video.raw`
  - `width` and `height`: dimensions in pixels of the images,
  - `pixel_format`: the way each pixel is encoded (such as `rgba` for red/green/blue/alpha or `yuva420p` as used in Liquidsoap, see `ffmpeg -pix_fmts`),
  - `pixel_aspect`: the aspect ratio of the image (typically `16:9` or `4:3`)
- `ffmpeg.audio.copy`: parameters are `codec` (the algorithm used to encode audio such as `mp3` or `aac`, see `ffmpeg -codecs` for a full list), `channel_layout`, `sample_format` and `sample_rate`,
- `ffmpeg.video.copy`: parameters are `codec`, `width`, `height`, `aspect_ratio` and `pixel_format`.

Passive and active sources. Most of the sources are *passive* which means that they are simply waiting to be asked for some data, they are not responsible for when the data is going to be produced. For instance, a playlist is a passive source: we can decode the files of the playlist at the rate we want, and will actually not decode any of those if we are not asked to. Similarly, the amplification operator `amplify(a, s)` is passive: it waits to be asked for data, then in turn asks the source `s` for data, and finally it returns the given data amplified by the coefficient `a`.

However, some sources are *active* which means that they are responsible for asking data. This is typically the case for outputs such as to a soundcard (e.g. `output.alsa`) or to a file (e.g. `output.file`). For instance, the (simplified) type of `output.alsa` is

```
(source(audio=pcm('a), video='b, midi='c)) -> active_source(audio=pcm('a),
↪  video='b, midi='c)
```

We see that it takes a source as input (the one to be played) and returns an active source: the returned source is the same as the input source, but the type indicates that it is active, as witnessed by the `active_source` instead of the usual `source`.

Perhaps surprisingly, some inputs are also tagged as active, because they are proactive. For instance, in the source `input.alsa`, we do not have control over the rate at which the data is produced, the soundcard sends us regularly audio data, and is responsible for the synchronization, and its type is

```
(...) -> active_source(audio=pcm('a), video='b, midi='c)
```

Any active source is a particular case of a source, so that we can feed the result of `input.alsa` to an operator requiring a source, such as `amplify` whose type is

```
({float}, source(audio=pcm('a), video='b, midi='c)) -> source(audio=pcm('a),
↪  video='b, midi='c)
```

as in the script

```
mic = input.alsa()
s   = amplify(2., mic)
output(buffer(s))
```

(namely, `mic` is of type `active_source` and `amplify` requires an argument of type `source`, which does not cause any problem).

This way of functioning means that if a source is not connected to an active source, its stream will not be produced. For instance, consider the following script:

```
s = playlist("~/Music")
s.on_track(fun(_) -> print("New track in the source!"))
output(blank())
```

Here, the only active source is `output` which is playing the `blank` source. The source `s` is not connected to an active source, and its contents will never be computed. This can be observed because we are printing a message for each new track: here, no stream is produced, thus no new track is produced, thus we will never see the message.

The above story is entirely not precise on one point. We will see in a section below that it is not the exactly the active sources themselves which are responsible for initiating computation of data, but rather the associated clocks.

Type inference. In order to determine the type of the sources, Liquidsoap looks where they are used and deduces constraints on their type. For instance, consider a script of the following form:

```
s = ...
output.alsa(s)
output.sdl(s)
```

In the first line, suppose that we do not know yet what the type of the source `s` should be. On the second line, we see that it is used as an argument of `output.alsa` and should therefore have a type of the form `source(audio=pcm('a), video='b, midi='c)`, i.e. the audio should be in `pcm` format. Similarly, on the third line, we see that it is used as an argument of `output.sdl` (which displays the video of the stream) and should therefore have a type of the form `source(audio='a, video=yuva420p('b), midi='c))`, i.e. the video should be in `yuva420p` format. Combining the two constraints, we deduce that the type of the source should be of the form `source(audio=pcm('a), video=yuva420p('b), midi='c)`.

In the end, the parameters of the stream which are not fixed will be taken to be default values. For instance, the number of audio channels will take the default value 2 (stereo), which is specified in the configuration option `frame.audio.channels`. If we want streams to be mono by default, we should type, at the beginning of the script,

```
set("frame.audio.channels", 1)
```

Similarly, the default number of MIDI channels is 0, since it is expected to be useless for most users, and can be changed in the configuration option `frame.midi.channels`. Once determined at startup, the contents of the streams (such as number of audio channels) is fixed during the whole execution of the script. Earlier versions of Liquidsoap somehow supported sources with varying contents, but this was removed because it turned out to be error-prone and not used much in practice.

During the type checking phase, it can happen that two constraints are not compatible for a given stream. In this case, an error is returned before the script is executed. For instance, suppose that we have a source `s` and we execute the following script:

```
t = amplify(3., s)
u = ffmpeg.decode.audio(t)
output(u)
```

We recall that the type of `amplify` is essentially

```
(float, source(audio=pcm('a), video='b, midi='c)) -> source(audio=pcm('a),
↪  video='b, midi='c)
```

and the one of `ffmpeg.decode.audio` is essentially

```
(source(audio=ffmpeg.audio.copy('a), video=none, midi=none)) ->
↪  source(audio=pcm('b), video=none, midi=none)
```

On the first line of the script above, we are using `amplify` on `s` which means that `s` should be of the form `source(audio=pcm('a), video='b, midi='c)`, i.e. the audio should be in `pcm` format, because `amplify` can only work on internal data. Moreover, the type of `t` should be the same as the one of `s` because the type of the output of `amplify` is the same as the source given as argument. However, on the second line, we use `u` as argument for `ffmpeg.decode.audio`, which means that it should have a type of the form `source(audio=ffmpeg.audio.copy('a), video=none, midi=none)` and now we have a problem: the audio of the source `u` should both be encoded in `pcm` and in `ffmpeg.audio.copy` formats, which is impossible. This explains why Liquidsoap raises the following error

```
At line 2, char 24:
Error 5: this value has type
  source(audio=pcm(_),...) (inferred at line 1, char 4-18)
but it should be a subtype of
  source(audio=ffmpeg.audio.copy(_),...)
```

which is a formal way of stating the above explanation.

Adding and removing channels. As a final remark on the design of our typing system, one could wonder why the type of the source returned by the `sine` operator is

```
source(audio=internal('a), video=internal('b), midi=internal('c))
```

and not

```
source(audio=internal('a), video=none, midi=none)
```

i.e. why allow the `sine` operator to generate video and MIDI data, whereas those are always quite useless (they are blank). The reason is mainly because of the following pattern. Suppose that you want to generate a blue screen with a sine wave as sound. You would immediately write something like this

```
a = sine()
b = video.fill(color=0x0000ff, blank())
s = add([a, b])
output.audio_video(s)
```

We create the source `a` which is the sine wave, the source `b` which is the blue screen (obtained by taking the output of `blank`, which is black and mute, and filling it in blue), we add them

and finally play the resulting source s. The thing is that we can only add sources of the same type: add being of type

```
([source(audio=internal('a), video=internal('b), midi=internal('c))]) ->
↪  source(audio=internal('a),video=internal('b),midi=internal('c))
```

it takes a list of sources to add, and lists cannot contain heterogeneous elements, otherwise said all the elements of a list should have the same type. Therefore, in order to produce a source with both audio and video, the elements of the list given as argument to add must all be sources with both audio and video.

If you insist on adding a video channel to a source which does not have one, you should use the dedicated function mux_video, whose type is

```
(video : source(audio=none, video='a, midi=none), source(audio='b, video=none,
↪  midi='c)) -> source(audio='b, video='a, midi='c)
```

(and the function mux_audio can similarly be used to add audio to a source which does not have that). However, since this function is much less well-known than add, we like to leave the possibility for the user to use both most of the time, as indicated above. Note however that the following variant of the above script

```
a = sine()
b = video.fill(color=0x0000ff, blank())
s = mux_video(video=b, a)
output.audio_video(mksafe(s))
```

is slightly more efficient since the source a does not need to generate video and the source b does not need to generate audio.

Dually, in order to remove the audio of a source, the operator drop_audio of type

```
(source(audio='a, video='b, midi='c)) -> source(audio=none, video='b, midi='c)
```

can be used, and similarly the operator drop_video can remove the video.

Type annotations. If you want to constrain the contents of a source, the Liquidsoap language offers the construction (e : t) which allows constraining an expression e to have type t (technically, this is called a type *cast*). It works for arbitrary expressions and types, but is mostly useful for sources. For instance, in the following example, we play the source s in mono, even though the default number of channels is two:

```
s = single("test.mp3")
output( (s : source(audio=pcm(mono))) )
```

Namely, in the second line, we constrain the type of s to be source(audio=pcm(mono)), i.e. a source with mono audio.

Encoding formats. In order to specify the format in which a stream is encoded, Liquidsoap uses particular annotations called *encoders*, already presented in there. For instance, consider the output.file operator which stores a stream into a file: this operator needs to know the kind of file we want to produce. The (simplified) type of this operator is

```
(format('a), string, source('a)) -> active_source('a)
```

We see that the second argument is the name of the file and the third argument is the source we want to dump. The first argument is the encoding format, of type format('a). Observe that it takes a type variable 'a as argument, which is the same variable as the parameters

of the source taken as argument, and the parameters of the returned source: the format required for the intput source will depend on the chosen format.

Encoders. The encoding formats are given by *encoders*, whose name always begin with the "%" character and can take parameters: their exhaustive list is given in there. For instance, if we want to encode a source `s` in MP3 format, we are going to use the encoder `%mp3` and thus write something like

```
output.file(%mp3, "/tmp/backup.mp3", s)
```

If we have a look at the type of the encoder `%mp3`, we see that its type is

```
format(audio=pcm(stereo), video=none, midi=none)
```

which means that, in the above example, the source `s` will be of type

```
source(audio=pcm(stereo), video=none, midi=none)
```

and thus have to contain stereo PCM audio, no video and no MIDI. The encoders take various parameters. For instance, if we want to encode MP3 in mono, at a bitrate of 192 kbps, we can pass the parameters `mono` and `bitrate=192` as follows:

```
output.file(%mp3(mono, bitrate=192), "/tmp/backup.mp3", s)
```

Some of those parameters will have an influence on the type of the stream. For instance, if we pass `mono` as parameter, the type of the encoder becomes

```
format(audio=pcm(mono), video=none, midi=none)
```

and thus imposes that `s` should have mono audio.

Because it has such an influence on types, an encoder is not a value as any other in Liquidsoap, and specific restrictions have to be imposed. In particular, you cannot use variables or complex expressions in the parameters for the encoders. For instance, the following will not be accepted

```
b = 192
output.file(%mp3(mono, bitrate=b), "/tmp/backup.mp3", s)
```

because we are trying to use the variable `b` as value for the bitrate. This is sometimes annoying and might change in the future.

Encoded sources. As another example of the influence of encoders, suppose that we want to encode our whole music library as a long MP3. We would proceed in this way:

```
s = playlist(loop=false, "~/Music")
s = clock(sync="none", s)
output.file(fallible=true, on_stop=shutdown, %mp3, "/tmp/out.mp3", s)
```

The first line creates a `playlist` source which will read all our music files once, the second line ensures that we try to encode the files as fast as possible instead of performing this in realtime as explained in there, and the third line requires the encoding in MP3 of the resulting source, calling the `shutdown` function once the source is over, which will terminate the script.

If you try this at home, you will see that it takes quite some time, because the `playlist` operator has to decode all the files of the library into internal raw contents, and the `output.file` operator has to encode the stream in MP3, which is quite CPU hungry. If our music library already consists of MP3 files, it is much more efficient to avoid decoding and then reencoding the files. In order to do so, we can use the FFmpeg encoder, by replacing the last line with

```
fmt = %ffmpeg(format="mp3", %audio.copy)
output.file(fallible=true, on_stop=shutdown, fmt, "/tmp/music.mp3", s)
```

Here, the encoder `fmt` states that we want to use the FFmpeg library, in order to create MP3, from already encoded audio (`%audio.copy`). In this case, the source `s` will have the type

```
source(audio=ffmpeg.audio.copy, video=none, midi=none)
```

where the contents of the audio is already encoded. Because of this, the `playlist` operator will not try to decode the MP3 files, it will simply pass their data on, and the encoder in `output.file` will simply copy them in the output file, thus resulting in a much more efficient script. More details can be found in there.

8.2 Frames

At this point, we think that it is important to explain a bit how streams are handled "under the hood", even though you should never have to explicitly deal with this in practice. After parsing a script, Liquidsoap starts one or more streaming loops. Each streaming loop is responsible for creating audio data from the inputs, pass it through the various operators and, finally, send it to the outputs. Those streaming loops are animated by *clocks*: each operator is attached to such a clock, which ensures that data is produced regularly. This section details this way of functioning.

Frames. For performance reasons, the data contained in streams is generated in small chunks, that we call *frames* in Liquidsoap. The default size of a frame is controlled by the `frame.duration` setting whose default value is 0.04 second, i.e. 1/25 th of a second. This corresponds to 1764 audio samples and 1 video image with default settings. The actual duration is detailed at the beginning of the logs:

```
Frames last 0.04s = 1764 audio samples = 1 video samples = 1764 ticks.
```

Changing the size. The size of frames can be changed by instructions such as

```
set("frame.duration", 0.12)
```

Note that if you request a duration of 0.06 second, by

```
set("frame.duration", 0.06)
```

you will see that Liquidsoap actually selects a frame duration of 0.08 seconds:

```
Frames last 0.08s = 3528 audio samples = 2 video samples = 3528 ticks.
```

this is because the requested size is rounded up so that we can fit an integer number of samples and images (0.06 would have amounted to 1.5 image per frame).

Pulling frames. In a typical script, such as

```
s = sine()
s = amplify(0.5, s)
output.pulseaudio(s)
```

The active source is `output.pulseaudio`, and is responsible for the generation of frames. In practice, it waits for the soundcard to say: "hey, my internal buffer is almost empty, now is a good time to fill me in!". Each time this happens, and this occurs 25 times per second, the active source generates a *frame*, which is a buffer for audio (or video) data waiting to be

filled in, and passes it to the `amplify` source asking it to fill it in. In turn, it will pass it to the `sine` source, which will fill it with a sine, then the `amplify` source will modify its volume, and then the `output.pulseaudio` source will send it to the soundcard. Note that, for performance reasons, all the operators work directly on the same buffer.

Assumptions on frame size. The frame duration is always supposed to be "small" so that values can be considered to be constant over a frame. For this reason, and in order to gain performance, expressions are evaluated only once at the beginning of each frame. For instance, the following script plays music at a random volume:

```
s = playlist("~/Music")
s = amplify({random.float()}, s)
output(s)
```

In fact, the random number for the volume is only generated once for the whole frame. This can be heard if you try to run the above script by setting the frame duration to a "large" number such as 1 second:

```
set("frame.duration", 1.)
```

You should be able to clearly hear that volume changes only once every second. In practice, with the default duration of a frame, this cannot be noticed. It can be sometimes useful to increase it a bit (but not as much as 1 second) in order to improve the performance of scripts, at the cost of decreasing the precision of computed values.

Triggering computations on frames. It is possible to trigger a computation on every frame, with the `source.on_frame` operator, which takes in addition to a source, a function which is called every time a new frame is computed. For instance, the following script will increase the volume of the source `s` by 0.01 on every frame:

```
v = ref(0.)
s = source.on_frame(s, {v := !v + 0.01})
s = amplify({!v}, s)
```

The default size of a frame being 0.04 s, volume will progressively be increased by 0.25 each second.

Frame raw contents. Let us provide some more details about the way data is usually stored in those frames, when using raw internal contents, which is the case most of the time. Each frame has room for audio, video and MIDI data, the format of this data we now describe.

Audio. The raw audio contents is called `pcm` for *pulse-code modulation*. The signal is represented by a sequence of *samples*, one for each channel, which represent the amplitude of the signal at a given instant. Each sample is represented by floating point number, between -1 and 1, stored in double precision (using 64 bits, or 8 bytes). The samples are given regularly for each channel of the signal, by default 44100 times per seconds: this value is called the *sample rate* of the signal and is stored globally in the `frame.audio.samplerate` setting. This means that we can retrieve the value of the samplerate with

```
f = get(default=0, "frame.audio.samplerate")
```

and set it to another value such as 48000 with

```
set("frame.audio.samplerate", 48000)
```

although default samplerate of 44100 Hz is largely the most commonly in use.

Video. A video consists of a sequence images provided at regular interval. By default, these images are presented at the *frame rate* of 25 images per second, but this can be changed using the setting `frame.video.framerate` similarly as above. Each image consists of a rectangle of pixels: its default width and height are 1280 and 720 respectively (this corresponds to the resolution called 720p or *HD ready*, which features an aspect ratio of 16:9 as commonly found on television or computer screens), and those values can be changed through the settings `frame.video.width` and `frame.video.height`. For instance, *full HD* or *1080p* format would be achieved with

```
set("frame.video.width",  1980)
set("frame.video.height", 1080)
```

Each pixel has a color and a transparency, also sometimes called an *alpha channel*: this last parameter specifies how opaque the pixels is and is used when superimposing two images (the less opaque a pixel of the above image is, the more you will see the pixels below it). Traditionally, the color would be coded in RGB, consisting of the values for the intensity of the red, green and blue for each pixel. However, if we did things in this way, every pixel would take 4 bytes (1 byte for each color and 1 for transparency), which means 4×1280×720×25 bytes (= 87 Mb) of video per seconds, which is too much to handle in realtime for a standard computer. For this reason, instead using the RGB representation, we use the YUV representation consisting of one *luma* channel Y (roughly, the black and white component of the image) and two *chroma* channels U and V (roughly, the color part of the image represented as blueness and redness). Moreover, since the human eye is not very sensitive to chroma variations, we can be less precise for those and take the same U and V values for 4 neighboring pixels. This means that each pixel is now encoded by 2.5 bytes on average (1 for Y, ¼ for U, ¼ for V and 1 for alpha) and 1 second of typical video is down to a more reasonable 54 Mb per second. You should now understand why the internal contents for video is called `yuva420p` in source types.

MIDI. MIDI stands for *Musical Instrument Digital Interface* and is a (or, rather, *the*) standard for communicating between various digital instruments and devices. Liquidsoap mostly follows it and encodes data as lists of *events* together with the time (in ticks, relative to the beginning of the frame) they occur and the channel on which they occur. Each event can be "such note is starting to play at such velocity", "such note is stopping to play", "the value of such controller changed", etc.

Encoded contents. As indicated in there, the data present in frames is not always in the above format. Namely, Liquidsoap also has support for frames whose contents is stored either in a format supported by the FFmpeg library, which can consist of encoded streams (e.g. audio in the MP3 format).

Ticks. The time at which something occurs in a frame is measured in a custom unit which we call *ticks*. To avoid errors due to rounding, which tend to accumulate when performing computations with float numbers, we want to measure time with integers. The first natural choice would thus be to measure time in audio samples, since they have the highest rate, and in fact this is what is done with default settings: 1 tick = 1 audio sample = 1/44100 second. In this case, an image lasts 1/25 second = 44100/25 ticks = 1764 ticks.

However, if we change the video framerate to 24 images per second with

```
set("frame.video.framerate", 24)
```

we have difficulties measuring time with integers because an image now lasts 44100/24 samples = 1837.5 samples, which is not an integral number. In this case, Liquidsoap conventionally decides that 1 sample = 2 ticks, so that an image lasts 3675 ticks. Indeed, if you try the above, you will see in the logs

```
Using 44100Hz audio, 24Hz video, 88200Hz main.
```

which means that there are 44100 audio samples, 24 images and 88200 ticks per second. You will also see in the logs

```
Frames last 0.08s = 3675 audio samples = 2 video samples = 7350 ticks.
```

which means that a frame lasts 0.8 seconds and contains 8675 audio samples and 2 video samples, which corresponds to 7350 ticks. More generally, the number of ticks per second as the smallest number such that both an audio and a video sample lasts for an integer number of ticks.

Tracks and metadata. Each frame contains two additional arrays of data which are timed, in ticks relative to the beginning of the frame: breaks and metadata.

Tracks. It might happen that a source cannot entirely fill the current frame. For instance, in the case of a source playing one file once (e.g. using the operator `once`), where there are only 0.02 seconds of audio left whereas the frame lasts 0.04 seconds. We could have simply ignored this and filled the last 0.02 seconds with silence, but we are not like this at Liquidsoap, especially since even such a short period of a silence can clearly be heard. Don't believe us? You can try the following script which sets up the frame size to 0.02 seconds and then silences the audio for one frame every second:

```
video.frame.rate.set(50)
frame.duration.set(0.02)
s = playlist("~/Music")
s = amplify({ (s.time() mod 1. <= 0.02) ? 0. : 1. }, s)
output(s)
```

You should clearly be able to hear a tick every second if the played music files are loud enough. For this reason, if a source cannot fill the frame entirely, it indicates it by adding a *break*, which marks the position until where the frame has been filled. If the frame is not complete, it will try to fill the rest on the next iteration of filling frames.

Each filling operation is required to add exactly one break. In a typical execution, the break will be at the end of the frame. If this is not the case, this means that the source could not entirely fill the frame, and this is thus considered as a *track* boundary. In Liquidsoap, tracks are encoded as breaks in frames which are not at the end: this mechanism is typically used to mark the limit between two successive songs in a stream. In scripts, you can detect when a track occurs using the `on_track` method that all sources have, and you can insert track by using the method provided by the `insert_metadata` function.

Metadata. A frame can also contain *metadata* which are pairs of strings (e.g. `"artist"`, `"Alizée"` or `"title"`, `"Moi... Lolita"`, etc.) together with the position in the frame where they should be attached. Typically, this information is present in files (e.g. mp3 files contain metadata encoded in ID3 format) and are passed on into Liquidsoap streams (e.g. when using the `playlist` operator). They are also used by output operators such as `output.icecast` to provide information about the currently playing song to the listener. In scripts, you can trigger a function when metadata is present with `on_metadata`, transform the metadata with

`map_metadata` and add new metadata with `insert_metadata`. For instance, you can print the metadata contained in tracks:

```
s = playlist("~/Music")
def print_metadata(m)
  list.iter(fun (lv) -> print("- #{fst(lv)}: #{snd(lv)}"), m)
end
s.on_metadata(print_metadata)
output(s)
```

If you have a look at a typical stream, you will recognize the usual information you would expect (artist, title, album, year, etc.). But you should also notice that Liquidsoap adds internal information such as

- `filename`: the name of the file being played,
- `temporary`: whether the file is temporary, i.e. has been downloaded from the internet and should be deleted after having been played,
- `source`: the name of the source which has produced the stream,
- `kind`: the kind (i.e. the contents of audio, video and MIDI) of the stream,
- `on_air`: the time at which it has been put on air, i.e. first played.

These are added when resolving requests, as detailed below. In order to prevent internal information leaks (we do not want our listeners to know about our filenames for instance), the metadata are filtered before being sent to outputs: this is controlled by the `"encoder.encoder.export"` setting, which contain the list of metadata which will be exported, and whose default value is

```
["artist", "title", "album", "genre", "date", "tracknumber",
 "comment", "track", "year", "dj", "next"]
```

8.3 The streaming model

The stream generation workflow. When starting the script, Liquidsoap begins with a *creation phase* which instantiates each source and computes its parameters by propagating information from the sources it uses. The two main characteristics determined for each source are

- *fallibility*: we determine whether the source is fallible, i.e. might be unable to produce its stream at some point (this is detailed below),
- *clocks*: we determine whether the source is synchronized by using the CPU or has its own way of keeping synced, e.g. using the internal clock of a soundcard (this is also detailed below).

Lifecycle of a source. The standard lifecycle of a source is the following one:

- we first inform the source that we are going to use it (we also say that we *activate* it) by asking it to *get ready*, which triggers its initialization,
- then we repeatedly ask it for *frames*,
- and finally, when the script shuts down, we *leave* the source, indicating that we are not going to need it anymore.

The information always flows from outputs to inputs. For instance, in a simple script such as

```
s = playlist("~/Music")
s = amplify(0.8, s)
```

```
output(s)
```

at beginning Liquidsoap will ask the output to get ready, in turn the output will ask the amplification operator to get ready, which will in turn ask the playlist to get ready (and leaving would be performed similarly, as well as the computation of frames as explained above). Note that a given source might be asked multiple times to get ready, for instance if it is used by two outputs (typically, an Icecast output and an HLS output). The first time it is asked to get ready, the source *wakes up* at which point it sets up what it needs (and dually, the last time it is asked to leave, the source goes to *sleep* where it cleans up everything). Typically, an `input.http` source, will start polling the distant stream at wake up time, and stop at sleep time.

You can observe this in the logs (you need to set your log level to least 5): when a source wakes up it emits a message of the form

```
Source xxx gets up ...
```

and when it goes to sleep it emits

```
Source xxx gets down.
```

where `xxx` is the identifier of the source (which can be changed by passing an argument labeled `id` when creating the source). You can also determine whether a source has been woken up, by using the method `is_up` which is present for any source `s`: calling `s.is_up()` will return a boolean indicating whether the source `s` is up or not. For instance,

```
s = playlist("~/Music")
thread.run(delay=1., { print("Source s is up: #{s.is_up()}.") })
```

will print, after 1 second, whether the playlist source is up or not (in this example it will always be the case).

Computing the kind, again. When waking up, the source also determines its *kind*, that is the number and nature of `audio`, `video` and `midi` channels as presented above. This might seem surprising because this information is already present in the type of sources, as explained above. However, for efficiency reasons, we drop types during execution, which means that we do not have access to this and have to compute it again (this is only done once at startup and is quite inexpensive anyway): some sources need this information in order to know in which format they should generate the stream or decode data. The computation of the kind is performed in two phases: we first determine the *content kind* which are the necessary constraints (e.g. we need at least one channel of PCM audio), and then we determine the *content type* where all the contents are fixed (e.g. we need two channels of PCM audio). When a source gets up it displays in the logs the requested content kind for the output, e.g.

```
Source xxx gets up with content kind: {audio=pcm,video=internal,midi=internal}.
```

which states that the source will produce PCM audio (but without specifying the number of channels), and video and MIDI in internal format. Later on, you can see lines such as

```
Content kind: {audio=pcm,video=internal,midi=internal},
content type: {audio=pcm(stereo),video=none,midi=none}
```

which mean that the content kind is the one described above and that the content type has been fixed to two channels of PCM audio, no video nor MIDI.

The streaming loop. As explained above, once the initialization phase is over, the outputs regularly ask the sources they should play to fill in frames: this is called the *streaming loop*. Typically, in a script of the form

```
s = switch([({6h-10h},morning), ({true},default)])
s = amplify(0.8, s)
output.icecast(%mp3, mount="...", s)
```

the Icecast output will ask the amplification operator to fill in a frame, which will trigger the switch to fill in a frame, which will require either the `morning` or `default` source to produce a frame depending on the time. For performance reasons, we want to avoid copies of data, and the computations are performed in place, which means that each operator directly modifies the frame produced by its source, e.g. the amplification operator directly changes the volume in the frame produced by the switch.

Since the computation of frames is triggered by outputs, when a source is shared by two outputs, at each round it will be asked twice to fill a frame (once by each source). For instance, consider the following script:

```
s = amplify(0.5, sine())
output.pulseaudio(s)
output.icecast(%mp3, mount="stream", s)
```

Here, the source `s` is used twice: once by the PulseAudio output and once by the Icecast output. Liquidsoap detects such cases and goes into *caching mode*: when the first active source (say, `output.pulseaudio`) asks `amplify` to fill in a frame, Liquidsoap will temporarily store the result (we say that it "caches" it, in what we call a *memo* frame) so that when the second active source asks `amplify` to fill in the frame, the stored one will be reused, thus avoiding computing twice a frame which would be disastrous (each output would have one frame every two computed frames).

Fallibility. Some sources can *fail*, which means that they do not have a sensible stream to produce at some point. This typically happens after ending a track when there is no more track to play. For instance, the following source `s` will play the file `test.mp3` once:

```
s = once(single("test.mp3"))
```

After the file has been played, there is nothing to play and the source fails. Internally, each source has a method to indicate whether it *is ready*, i.e. whether it has something to play. Typically, this information is used by the `fallback` operator in order to play the first source which is ready. For instance, the following source will try to play the source `s`, or a sine if `s` is not ready:

```
t = fallback([s, sine()])
```

In Liquidsoap scripts, every source has a method `is_ready` which can be used to determined whether it has something to play.

On startup, Liquidsoap ensures that the sources used in outputs never fail (unless the parameter `fallible=true` is passed to the output). This is done by propagating fallibility information from sources to sources. For instance, we know that a `blank` source or a `single` source will never fail (for the latter, this is because we download the requested file at startup), `input.http` is always fallible because the network might go down, a source `amplify(s)` has the same fallibility as `s`, and so on. Typically, if you try to execute the script

```
s = input.http("http://...")
output.pulseaudio(s)
```

Liquidsoap will issue the error

```
Error 7: Invalid value: That source is fallible
```

indicating that it has determined that we are trying to play the source `s`, which might fail. The way to fix this is to use the `fallback` operator in order to play a file which is always going to be available in case `s` falls down:

```
s         = input.http("http://...")
emergency = single("/radio/emergency.mp3")
s         = fallback(track_sensitive=false, [s, emergency])
output.pulseaudio(s)
```

Or to use `mksafe` which is defined by

```
def mksafe(s)
  fallback(track_sensitive=false, [s, blank()])
end
```

and will play blank in case the input source is down.

The "worse" source with respect to fallibility is given by the operator `source.fail`, which creates a source which is never ready. This is sometimes useful in order to code elaborate operators. For instance, the operator `once` is defined from the `sequence` operator (which plays one track from each source in a list) by

```
def once(s)
  sequence([s, source.fail()])
end
```

Another operator which is related to fallibility is `max_duration` which makes a source unavailable after some fixed amount of time.

Clocks. Every source is attached to a particular a *clock*, which is fixed during the whole execution of the script, and is responsible for determining when the next frame should be computed: at regular intervals, the clock will ask active sources it controls to generate frames. We have said that a frame lasts for 0.04 seconds by default, which means that a new frame should be computed every 0.04 seconds, or 25 times per second. The clock is responsible for measuring the time so that this happens at the right rate.

Multiple clocks. The first reason why there can be multiple clocks is *external*: there is simply no such thing as a canonical 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, and each sound library makes a different use of the soundcard. For applications such as radios, which are supposed to run for a very long time, this is a problem. A discrepancy of 1 millisecond every second will accumulate to a difference of 43 minutes after a month: this means that at some point in the month we will have to insert 43 minutes of silence or cut 43 minutes of music in order to synchronize back the two clocks! The use of clocks allows Liquidsoap to detect such situations and require the user to deal with them. In practice, this means that each library (ALSA, Pulseaudio, etc.) has to be attached to its own clock, as well as network libraries taking care of synchronization by themselves (SRT).

There are also some reasons that are purely *internal* to Liquidsoap: in order to produce a stream at a given rate, 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, such as `stretch`. But it also holds more subtly for operators such as `cross`, which is responsible for crossfading successive tracks in a source: during the lapse of time where the operator combines data from an end of track with the beginning of the next one, 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.

The use of clocks in Liquidsoap ensures that a given source will not be pulled at two different rates by two operators. This guarantees that each source will only have to sequentially produce data and never simultaneously produce data for two different logical instants, which would be a nightmare to implement correctly.

Observing clocks. Consider the following script:

```
s = input.alsa()
s = amplify(0.8, s)
output.file(%mp3, "/tmp/out.mp3", s)
```

Here, the only operator to enforce the use of a particular clock is `input.alsa` and therefore its clock will be used for all the operators. Namely, we can observe in the logs that `input.alsa` uses the `alsa` clock

```
[input.alsa_64610:5] Clock is alsa[].
```

and that the `amplify` operator is also using this clock

```
[amplify_64641:5] Clock is alsa[].
```

Once all the operators created and initialized, the clock will start its *streaming loop* (i.e. produce a frame, wait for some time, produce another frame, wait for some time, and so on):

```
[clock.alsa:3] Streaming loop starts in auto-sync mode
```

Here, we can see that ALSA is taking care of the synchronization, this is indicated by the message:

```
[clock.alsa:3] Delegating synchronisation to active sources
```

If we now consider a script where there is no source which enforces synchronization such as

```
s = sine()
output.file(%mp3, "/tmp/out.mp3", s)
```

we can see in the logs that the CPU clock, which is called `main`, is used

```
[sine_64611:5] Clock is main[].
```

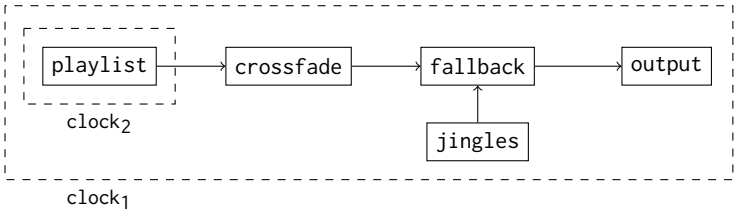
and that synchronization is taken care of by the CPU

```
[clock.main:3] Delegating synchronisation to CPU clock
```

Graphical representation. In case it helps to visualize clocks, a script can be drawn as some sort of graph whose vertices are the operators and there is an arrow from a vertex `op` to a vertex `op'` when the operator `op'` uses the stream produced by the operator `op`. For instance, a script such as

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

can be represented as the following graph:



The dotted boxes on this graph represent clocks: all the nodes in a box are operators which belong to the same clock. Here, we see that the `playlist` operator has to be in its own clock $\texttt{clock}_2$, because it can be manipulated in a non-linear way by the `crossfade` operator in order to compute transitions, whereas all other operators belong the same clock $\texttt{clock}_1$ and will produce their stream at the same rate.

Errors with clocks. At startup Liquidsoap assigns a clock to each operator by applying the three following rules:

1. we should follow the clock imposed by operators which have special requirements:
   - `input.alsa` and `output.alsa` have to be in the `alsa` clock, `input.pulseaudio` and `output.pulseaudio` have to be in the `pulseaudio` clock, etc.,
   - the sources used by `stretch`, `cross` and few other "time-sensitive" operators have their own clock,
   - the operator `clock` generates a new clock,
2. each operator should have the same clock as the sources it is using (unless for special operators such as `cross` or `buffer`): this called clock *unification*,
3. if the two above rules do not impose a clock to an operator, it is assigned to the default clock `main`, which based on CPU.

It should always be the case that a given operator belongs to exactly one clock. If, by applying the above rules, we discover that an operator should belong to two (or more) clocks, we raise an error. For instance, the script

```
output.alsa(s)
output.pulseaudio(s)
```

will raise at startup the error

```
A source cannot belong to two clocks (alsa[], pulseaudio[]).
```

because the source `s` should be both in the `alsa` and in the `pulseaudio` clock, which is forbidden. This is for a good reason: the ALSA and the Pulseaudio libraries each have their own way of synchronizing streams and might lead to the source `s` being pulled at two different rates. Similarly, the script

```
o = add([stretch(ratio=2., s), s])
```

will raise the error

```
Cannot unify two nested clocks (resample_65223[],
↪  ?(3f894ac2d35c:0)[resample_65223[]]).
```

because the source `s` should belong to the clock used by `stretch` and the clock of `stretch`. When we think about it the reason should be clear: we are trying to add the source `s` played at normal speed and at a speed slowed down twice. This means that in order to compute

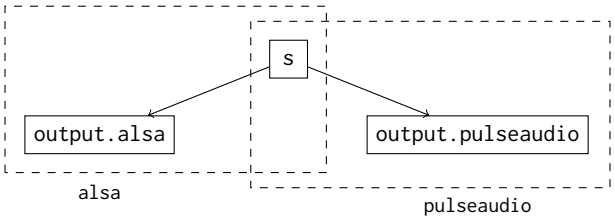
the stream o at a given time t, we need to know the stream s both at time t and at time $t/2$, which is forbidden because we only want to compute a source at one logical instant.

Mediating between clocks: buffers. As we have seen in there, the usual way to handle clock problems is to use buffer operators (either `buffer` of `buffer.adaptative`): they record in a buffer some of their input source before outputting it (1 second by default), so that it can easily cope with small time discrepancies. Because of this, we allow that the clock of their argument and their clocks are different.

For instance, we have seen that the script

```
output.alsa(s)
output.pulseaudio(s)
```

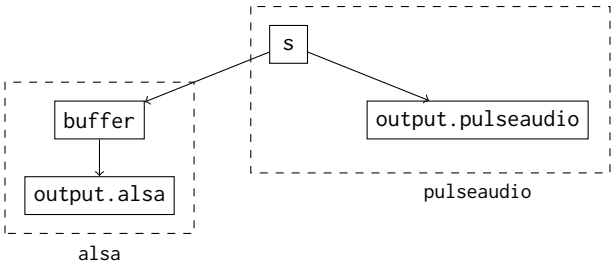
is not allowed because it would require `s` to belong to two distinct clocks. Graphically,



The easy way to solve this is to insert a `buffer` operator before one of the two outputs, say `output.alsa`:

```
output.alsa(fallible=true, buffer(s))
output.pulseaudio(s)
```

which allows having two distinct clocks at the input and the output of `buffer` and thus two distinct clocks for the whole script:



Catching up. We have indicated that, by default, a frame is computed every 0.04 second. In some situations, the generation of the frame could take more than this: for instance, we might fetch the stream over the internet and there might be a problem with the connection, or we are using very CPU intensive audio effects, and so on. What happens in this case? If this is for a very short period of time, nothing: there are buffers at various places, which store the stream in advance in order to cope with this kind of problems. If the situation persists, those buffer will empty and we will run into trouble: there is not enough audio data to play and we will regularly hear no sound.

This can be tested with the `sleeper` operator, which can be used to simulate various audio delays. Namely, the following script simulates a source which takes roughly 1.1 second to

generate 1 second of sound:

```
s = sleeper(delay=1.1, sine())
output(s)
```

When playing it you should hear regular glitches and see messages such as

```
2020/07/29 11:13:05 [clock.pulseaudio:2] We must catchup 0.86 seconds!
```

This means Liquidsoap took n+0.86 seconds to produce n seconds of audio, and is thus "late". In such a situation, it will try to produce audio faster than realtime in order to "catch up" the delay.

Coping witch catch up errors. How can we cope with this kind of situations? Again, buffers are a solution to handle temporary disturbances in production of streams for sources. You can explicitly add some in you script by using the `buffer` operator: for instance, in the above script, we would add before the output, the line

```
s = buffer(s)
```

which make the source store 1 second of audio (this duration can be configured with the `buffer` parameter) and thus bear with delays of less than 1 second.

A more satisfactory way to fix this consists in identifying the cause of the delay, but we cannot provide a general answer for this, since it largely depends on your particular script. The only general comment we can make is that something is taking time to compute at some point. It could be that your CPU is overloaded and you should reduce the number of effects, streams or simultaneous encodings. It could also come from the fact that you are performing operations such as requests over the internet, which typically take time. For instance, we have seen in an earlier section that we can send the metadata of each track to a website with a script such as

```
radio = playlist("~/Music")
def handle_metadata(m)
  _ = http.post(data=json.stringify(m),
        headers=[("Content-Type", "application/json; charset=UTF-8")],
        "http://our.website.com/update_metadata.php")
end
radio.on_track(handle_metadata)
```

which uses `http.post` to POST the metadata of each track to a distant server. Since the connection to the server can take time, it is much better to perform it in a separate thread, which will run in parallel to the computation of the stream, without inducing delay in it. This can be performed by calling the `handle_metadata` with the `thread.run` function, i.e. replace the last line above by

```
radio.on_track(fun(m) -> thread.run({handle_metadata(m)}))
```

A last way of dealing with the situation is by simply ignoring it. If the only thing which is disturbing you is the error messages that pollute your log and not the error itself, you can have fewer messages by changing the `clock.log_delay` setting which controls how often the "catchup" error message is displayed. For instance, with

```
set("clock.log_delay", 60.)
```

you will only see one every minute.

8.4 Requests

When passing something to play to an operator, such as `test.mp3` to the operator `single`,

```
s = single("file.mp3")
```

it seems that the operator can simply open the file and play it on the go. However, things are a bit more complicated in practice. Firstly, we have to actually get the file:

- the file might be a distant file (e.g. `http://some.server/file.mp3` or `ftp://some.server/file.mp3`), in which case we want to download it beforehand in order to ensure that we have a valid file and that we will not be affected by the network,
- the "file" might actually be more like a recipe to produce the file (for instance `say:Hello you`, means that we should take some text-to-speech program to generate a sound file with the text `Hello you`).

Secondly, we have to find a way to decode the file

- we have to guess what format it is, based on the header of the file and its extension,
- we have to make sure that the file is valid and find a *decoder*, i.e. some library that we support which is able to decode it,
- we have to read the metadata of the file

Finally, we have to perform some cleanup after the file has been played:

- the decoder should be cleanly stopped,
- temporary files (such as downloaded files) have to be removed.

Also note that the decoder depends on the kind of source we want to produce: for instance, an MP3 file will not be acceptable if we are trying to generate video, but will of course be if we are trying to produce audio only.

For those reasons, most operators (such as `single`, `playlist`, etc.) do not directly deal with files, but rather with *requests*. Namely, a request is an abstraction which allows manipulating files but also performing the above operations.

Requests. A *request* is something from which we can eventually produce a file.

URI. It starts with an URI (*Uniform Resource Identifier*), such as

- `/path/to/file.mp3`
- `http://some.server/file.mp3`
- `annotate:title="My song",artist="The artist":~/myfile.mp3`
- `replaygain:/some/file.mp3`
- `say:This is my song`
- `synth:shape=sine,frequency=440.,duration=10.`
- ...

As you can see the URI is far from always being the path to a file. The part before the first colons (`:`) is the *protocol* and is used to determine how to fetch or produce the file. A local file is assumed when no protocol is specified. Some protocols such as `annotate` or `replaygain` operate on URI, which means that they allow chaining of protocols so that

```
replaygain:annotate:title="Welcome":say:Hello everybody!
```

is a valid request.

The status of a request. When a request is created it is assigned a RID, for *request identifier*, which is a number which uniquely identifies it (in practice the first request has RID 0, the second one RID 1, and so on). Each request also has a *status* which indicate where it is in its lifecycle:

1. *idle*: this is the initial status of a request which was just created,
2. *resolving*: we are generating an actual file for the request,
3. *read*: the request is ready to be played,
4. *playing*: the request is currently being played by an operator,
5. *destroyed*: the request has been played and destroyed (it should not be used anymore).

Resolution. The process of generating a file from a request is called *resolving* the request. The *protocol* specifies the details of this process, which is done in two steps:

1. some computations are performed (e.g. sound in produced by a text-to-speech library for `say`),
2. a list of URI, called *indicators*, is returned.

Generally, only one URI is returned: for instance, the `say` protocol generates audio in a temporary file and returns the path to the file it produced. When multiple URI are returned, Liquidsoap is free to pick any of them and will actually pick the first working one. Typically, a "database" protocol could return multiple locations of a given file on multiple servers for increased resiliency.

When a request is indicated as *persistent* is can be played multiple times (this is typically the case for local files). Otherwise, a request should only be used once. Internally, with every indicator is also associated the information of whether it is *temporary* or not. If it is, the file is removed when the request is destroyed. For instance, the `say` protocol generates the text in a temporary file, which we do not need after it has been played.

When resolving the request, after a file has been generated, Liquidsoap also ensures basic checks on data and computes associated information:

- we read the metadata in the file (and convert those to the standard UTF-8 encoding for characters),
- we find a library to decode the file (a decoder).

The resolution of a request may *fail* if the protocol did not manage to successfully generate a file (for instance, a database protocol used with a query which did not return any result) or if no decoder could be found (either the data is invalid or the format is not supported).

Manipulating requests. Requests can be manipulated within the language with the following functions.

- `request.create` creates a request from an URI. It can be specified to be persistent or temporary with the associated arguments. Beware that temporary files are removed after they have been played so that you should use this with care.
- `request.resolve` forces the resolution of a request. This function returns a boolean indicating whether the resolution succeeded or not. The `timeout` argument specifies how much time we should wait before aborting (resolution can take long, for instance when downloading a large file from a distant server). The `content_type` argument indicates a source with the same content type (number and kind of audio and video channels) as the source for which we would like to play the request: the resolution depends on it (for instance, we cannot decode an MP3 file to produce video…). Resolving twice does not hurt: resolution will simply not do anything the second time.

- `request.destroy` indicates that the request will not be used anymore and associated resources can be freed (typically, we remove temporary files).
- `request.id` returns the RID of the request.
- `request.status` returns the current status of a request (idle, resolving, ready, playing or destroyed) and `request.ready` indicates whether a request is ready to play.
- `request.uri` returns the initial URI which was used to create the request and `request.filename` returns the file to which the request resolved.
- `request.duration` returns the (estimated) duration of the request in seconds.
- `request.metadata` returns the metadata associated to request. This metadata is automatically read when resolving the file with a specified content type. The function `request.read_metadata` can be used to force reading the metadata in the case we have a local file.
- `request.log` returns the log associated to a particular request. It is useful in order to understand why a request failed to resolve and can also be obtained by using the `request.trace` telnet command.

Requests can be played using operators such as

- `request.queue` which plays a dynamic queue of requests,
- `request.dynamic` which plays a sequence of dynamically generated requests,
- `request.once` which plays a request once.

Those operators take care of resolving the requests before using them and destroying them afterward.

Metadata. When resolving requests, Liquidsoap inserts metadata in addition to the metadata already contained in the files. This can be observed with the following script:

```
r = request.create("test.mp3")
print(request.metadata(r))
```

Here, we are creating a request from a file path `test.mp3`. Since we did not resolve the request, the metadata of the file has not been read yet. However, the request still contains metadata indicating internal information about it. Namely, the script prints:

```
[("filename", "test.mp3"), ("temporary", "false"),
 ("initial_uri", "test.mp3"), ("status", "idle"), ("rid", "0")]
```

The meaning of the metadata should be obvious:

- `rid` is the identifier of the request,
- `status` is the status of the request,
- `initial_uri` is the URI we used to create the request,
- `filename` is the file the request resolved to (here, already had a local file so that it does not change)
- `temporary` indicates whether the file is temporary or not.

Protocols. The list of protocols available in Liquidsoap for resolving requests can be obtained by typing the command

```
liquidsoap --list-protocols-md
```

on on the website. The documentation also indicates which protocol are *static*: for those, the same URI should always produce the same result, and Liquidsoap can use this information in order to optimize the resolution.

Some of those protocols are built in the language such as

- `http` and `https` to download distant files over HTTP,
- `annotate` to add metadata.

Some other protocols are defined in the standard library (in the file `protocols.liq`) using the `add_protocol` function which registers a new protocol. This function takes as argument a function `proto` of type

```
(rlog : ((string) -> unit), maxtime : float, string) -> [string]
```

which indicates how to perform the resolution: this function takes as arguments

- `rlog` a function to write in the request's log,
- `maxtime` the maximal duration resolution should take,
- the URI to resolve,

and returns a list of URI it resolves to. Additionally, the function `add_protocol` takes arguments to document the function (`syntax` describes the URI accepted by this protocol and `doc` is freeform description of the protocol) as well as indicate whether the protocol is `static` or not and whether the files it produces are `temporary` or not.

Request leaks. At any time, a given script should only have a few requests alive. For instance, a `playlist` operator has a request for the currently playing file and perhaps for a few files in advance, but certainly not for the whole playlist: if the playlist contained distant files, this would mean that we would have to download them all before starting to play. Because of this, Liquidsoap warns you when there are hundreds of requests alive: this either mean that you are constantly creating requests, or that they are not properly destroyed (what we call a *request leak*). For instance, the following script creates 250 requests at once:

```
def loop()
  for i = 1 to 250 do
    ignore(request.create("test.mp3"))
  end
end
thread.run(delay=1., loop)
```

Consequently, you will therefore see in the logs messages such as

```
2021/05/04 12:22:18 [request:2] There are currently 100 RIDs, possible request
↪  leak! Please check that you don't have a loop on empty/unavailable requests,
↪  or creating requests without destroying them. Decreasing request.grace_time
↪  can also help.
2021/05/04 12:22:18 [request:2] There are currently 200 RIDs, possible request
↪  leak! Please check that you don't have a loop on empty/unavailable requests,
↪  or creating requests without destroying them. Decreasing request.grace_time
↪  can also help.
```

Decoders. As mentioned above, the process of resolving requests involves finding an appropriate decoder.

Configuration. The list of available decoders can be obtained with the script

```
print(get(default=[], "decoder.decoders"))
```

which prints here

```
["WAV", "AIFF", "PCM/BASIC", "MIDI", "IMAGE", "RAW AUDIO", "FFMPEG", "FLAC",
↪  "AAC", "MP4", "OGG", "MAD", "GSTREAMER"]
```

indicating the available decoders. The choice of the decoder is performed on the MIME type (i.e. the detected type for the file) and the file extension. For each of the decoders the configuration key

- `decoder.mime_types.*` specifies the list of MIME types the decoder accepts,
- `decoder.file_extension.*` specifies the list of file extensions the decoder accepts.

For instance, for the mad decoder (mad is a library to decode MP3 files) we have

```
set("decoder.mime_types.mad", ["audio/mpeg","audio/MPA"])
set("decoder.file_extensions.mad", ["mp3","mp2","mp1"])
```

Finally, the configuration key `decoder.priorities.*` specify the priority of the decoder. For instance,

```
set("decoder.priorities.mad", 1)
```

The decoders with higher priorities are tried first, and the first decoder which accepts a file is chosen. For MP3 files, this means that the FFmpeg decoder is very likely to be used over mad, because it also accepts MP3 files but has priority 10 by default.

Custom decoders. It is possible to add your custom decoders using the `add_decoder` function, which registers an external program to decode some audio files: this program should read the data on standard input and write decoded audio in WAV format on its standard output.

The log of a resolution. The choice of a decoder can be observed when setting log level to debug. For instance, consider the simple script

```
set("log.level", 5)
output(single("test.mp3"))
```

We see the following steps in the logs:

- the source `single` decides to resolve the request `test.mp3`:

  ```
  [single_65193:3] "test.mp3" is static, resolving once for all...
  [single_65193:5] Content kind: {audio=pcm,video=any,midi=any}, content type:
  ↪  {audio=pcm(stereo),video=none,midi=none}
  [request:5] Resolving request [[test.mp3]].
  ```

- some decoders are discarded because the extension or the MIME are not among those they support:

  ```
  [decoder.ogg:4] Invalid file extension for "test.mp3"!
  [decoder.ogg:4] Invalid MIME type for "test.mp3": audio/mpeg!
  [decoder.mp4:4] Invalid file extension for "test.mp3"!
  [decoder.mp4:4] Invalid MIME type for "test.mp3": audio/mpeg!
  [decoder.aac:4] Invalid file extension for "test.mp3"!
  [decoder.aac:4] Invalid MIME type for "test.mp3": audio/mpeg!
  [decoder.flac:4] Invalid file extension for "test.mp3"!
  [decoder.flac:4] Invalid MIME type for "test.mp3": audio/mpeg!
  [decoder.aiff:4] Invalid file extension for "test.mp3"!
  [decoder.aiff:4] Invalid MIME type for "test.mp3": audio/mpeg!
  ```

```
[decoder.wav:4] Invalid file extension for "test.mp3"!
[decoder.wav:4] Invalid MIME type for "test.mp3": audio/mpeg!
```

- two possible decoders are found, FFmpeg and mad, the first one having priority 10 and the second one priority 1

  ```
  [decoder:4] Available decoders: FFMPEG (priority: 10), MAD (priority: 1)
  ```

- the one with the highest priority is tried first, accepts the file, and is thus selected:

  ```
  [decoder.ffmpeg:4] ffmpeg recognizes "test.mp3" as: audio: {codec: mp3,
  ↪  48000Hz, 2 channel(s)} and content-type:
  ↪  {audio=pcm(stereo),video=none,midi=none}.
  [decoder:4] Selected decoder FFMPEG for file "test.mp3" with expected kind
  ↪  {audio=pcm(stereo),video=none,midi=none} and detected content
  ↪  {audio=pcm(stereo),video=none,midi=none}
  ```

- the resolution process is over:

  ```
  [request:5] Resolved to [[test.mp3]].
  ```

The log of a failed resolution. For comparison, consider the following variant of the script

```
set("log.level", 5)
output.video(single("test.mp3"))
```

Here, the resolution will fail because we are trying to play the source with `output.audio_video`: this implies that the source should have video, which an MP3 does not. The logs of the resolution process are as follows:

- the source `single` initiates the resolution of `test.mp3`:

  ```
  [single_65193:3] "test.mp3" is static, resolving once for all...
  [single_65193:5] Content kind: {audio=any,video=yuva420p,midi=any}, content
  ↪  type: {audio=pcm(stereo),video=yuva420p,midi=none}
  [request:5] Resolving request [[test.mp3]].
  ```

  You can observe that the content type has `audio=pcm(stereo)`, which means that we want stereo audio and `video=yuva420p` which means that we want video,

- some decoders are discarded because the extension or MIME is not supported:

  ```
  [decoder.ogg:4] Invalid file extension for "test.mp3"!
  [decoder.ogg:4] Invalid MIME type for "test.mp3": audio/mpeg!
  ```

- the FFmpeg decoder is tried (mad is not considered because it cannot produce video):

  ```
  [decoder:4] Available decoders: FFMPEG (priority: 10)
  [decoder.ffmpeg:4] ffmpeg recognizes "test.mp3" as: audio: {codec: mp3,
  ↪  48000Hz, 2 channel(s)} and content-type:
  ↪  {audio=pcm(stereo),video=none,midi=none}.
  [decoder:4] Cannot decode file "test.mp3" with decoder FFMPEG. Detected
  ↪  content: {audio=pcm(stereo),video=none,midi=none}
  ```

  we see that the decoder detects that the contents of the file is stereo audio and no audio, consequently it refuses to decode the file because we are requesting video,

- not decoder was found for the file at the given content type and the resolution process fails (an empty list of indicators is returned):

```
[decoder:3] Available decoders cannot decode "test.mp3" as
↪  {audio=pcm(stereo),video=yuva420p,midi=none}
[request:5] Resolved to [].
```

- the `single` operator raises a fatal exception because it could not resolved the URI we asked for:

```
[clock.main:4] Error when starting graphics:
↪  Request_simple.Invalid_URI("test.mp3")!
```

Other libraries involved in the decoding of files. Apart from decoders, the following additional libraries are involved when resolving and decoding requests.

- *Metadata decoders*: some decoders are dedicated to decoding the metadata of the files.
- *Duration decoders*: some decoders are dedicated to computing the duration of the files. Those are not enabled by default and can be by setting the dedicated configuration key

  ```
  set("request.metadata_decoders.duration", true)
  ```

  The reason they are not enabled is that they can take quite some time to compute the duration of a file. If you need this, it is rather advised to precompute it and store the result in the `duration` metadata.
- *Samplerate converters*: those are libraries used to change the samplerate of audio files when needed (e.g. converting files sampled at 48 kHz to default 44.1 kHz). The following configuration key sets the list of converters:

  ```
  set("audio.converter.samplerate.converters",
      ["ffmpeg","libsamplerate","native"])
  ```

  The first supported one is chosen. The `native` converter is fast and always available, but its quality it not very good (correctly resampling audio is a quite involved process), so that we recommend that you compile Liquidsoap with FFmpeg or libsamplerate support.
- *Channel layout converters*: those convert between the supported audio channel layouts (currently supported are mono, stereo and 5.1). Their order can be changed with the `audio.converter.channel_layout.converters` configuration key.
- *Video converters*: those convert between various video formats. The converter to use can be changed by setting the `video.converter.preferred` configuration key.

Custom metadata decoders can be added with the function `add_metadata_resolver`.

8.5 Reading the source code

As indicated in there, a great way of learning about Liquidsoap, and adding features to it, is to read (and modify) the standard library, which is written in the Liquidsoap language detailed in the dedicated chapter. However, in the case you need to modify the internal behavior of Liquidsoap or chase an intricate bug you might have to read (and modify) the code of Liquidsoap itself, which is written in the OCaml language. This can be a bit intimidating at first, but it is perfectly doable with some motivation, and it might be reassuring to learn that some other people have gone through this before you!

In order to guide you through the source, let us briefly describe the main folders and files. All the files referred to here are in the `src` directory of the source, where all the code lies. The main folders are

- language:
  - `lang/`: the definition of language and all the builtin operators,
  - `stream/`: internal representation and manipulation of streams using frames,
- operators:
  - `operators/`: where most operators such as sound processing are,
  - `conversions/`: conversion operators such as `mean`, `drop_audio`, `mux_audio`, etc.
- inputs and outputs:
  - `io/`: libraries performing both input and output such as ALSA,
  - `sources/`: input sources
  - `outputs/`: outputs,
- file formats:
  - `decoder/`, `encoder_formats/` and `encoder/`: decoders and encoders using various libraries for various formats and codecs,
  - `converters`: audio samplerate and image formats converters,
  - `lang_encoders`: support in the language for various encoders,
- protocols: `protocols/`.

The most important files are the following ones:

File	Description
`lang/lang_types.ml`	Types of the language
`lang/lang_values.ml`	Expressions of the language
`lang/lang.ml`	High-level operations on the language
`stream/frame.ml`	Definition of frames for streams
`sources.ml`	Definition of sources
`clock.ml`	Definition of clocks
`request.ml`	Definition of requests

Happy hacking, and remember that the community is here to help you!

Bibliography

Baelde, David, Romain Beauxis, and Samuel Mimram. 2011. "Liquidsoap: A High-Level Programming Language for Multimedia Streaming." In *International Conference on Current Trends in Theory and Practice of Computer Science*, 99–110. Springer. https://arxiv.org/abs/1104.2681.

Baelde, David, and Samuel Mimram. 2008. "De la webradio lambda à la λ-webradio." In *JFLA (Journées Francophones des Langages Applicatifs)*, 47–62. Étretat, France. http://hal.inria.fr/inria-00202813.