# LASH Audio Session Handler: Past, Present, and Future

**Juuso ALASUUTARI**
LASH development team
http://lash-audio.org/
Juuso.Alasuutari@gmail.com

**Abstract**

The LASH Audio Session Handler is a framework for saving and recalling the combined state of a collection of running audio programs. It is a central part of the Linux audio desktop, and its success or failure may well shape the way Linux audio software is received by a larger demography of users. During the past year, LASH has become more tightly integrated with modern desktop environments and continues to evolve in this direction.

**Keywords**

LASH, desktop, integration, usability.

## 1 Introduction

The LASH Audio Session Handler is arguably one of the central components of the Linux audio desktop. In principle, the concept behind LASH can, when properly realized, open the doors to widespread adoption of Free/Open Source audio software on many fronts.

LASH establishes a common framework for saving and recalling the combined state of a collection of running audio programs [1]. That is, with LASH (and a proper front-end) the user can save and restore his/her audio desktop state with a few mouse clicks.

After a period of stagnation the LASH project is again making active progress towards the 1.0 milestone.

## 2 How LASH Works

LASH comprises two parts; the system service and the client library. In this respect it is quite similar to JACK [2].

For the client programmer, making use of LASH amounts to implementing certain calls and functions in the application and linking it against liblash.so. Making an application LASH-aware also comes with some requirements with respect to how the program should behave.

On start-up a LASH-aware application should contact the LASH service and provide it with some information about itself. After this it is forbidden of the application to act on its own with respect to manipulating its state; instead, LASH will send it commands which reflect the user's decisions.

By relinquishing certain aspects of their autonomy and submitting to the LASH service, audio applications can be choreographed to form a coherent, desktop-wide session manageable via common front-ends.

### 2.1 Highlights of the Current API

The current LASH API (found in LASH 0.5.4) is based on receiving and sending events.

A program using the current stable API must pass its command-line arguments to lash_extract_args(), which parses them for any relevant LASH settings. It then returns an object that must be passed to lash_init(), which in turn initializes the connection to the LASH daemon.

After lash_init() the client program must inform LASH of its desired client name, its JACK client name, and its ALSA client ID, if appropriate. After these steps it becomes possible for the client to check for, receive, and send back event objects.

The current API is scheduled to be replaced by a more modern, callback-based one by the release of LASH 1.0.

## 3 A Common LASH Use Case

In the most common LASH use case a user wishes to save his or her work after using one or more audio applications to compose a piece of music. The user might have needed a sequencer, a software synthesizer, and a drum machine for the task.

In the above case, given that all relevant applications are LASH-aware and that there exists a capable LASH front-end sitting in the system tray, the user may save the project by clicking the front-end's icon and selecting “Save” from a pop-up menu. A progress bar may appear, and the user

will receive a notification when the save is complete.
At a later time, perhaps after a reboot, the user may select to reload the project he/she was working on earlier. Again after clicking on the front-end's icon, the user can navigate to a menu that lists all previously saved projects, right-click above the desired one, and select “Load”. LASH will then launch the relevant audio applications, instruct them to load their previously saved settings, and reconnect their JACK and ALSA ports accordingly.

## 4 Recent LASH Developments

Much has been going on in the LASH development repository since the latest stable version, 0.5.4, was released. The following is a compilation of the most noteworthy developments to date.

### 4.1 Summercode 2008

The LASH project received a stroke of fortune in the form of a sponsorship by the Finnish Centre for Open Source Solutions (COSS) in 2008 [3].

COSS organizes an annual, Google Summer of Code-type event called Summercode where students offer themselves to work on a project of their choice for three months. The author of this paper, Juuso Alasuutari was among the five people selected, and consequently he spent the spring and summer making several changes and adding new features to LASH.

### 4.2 D-Bus Connectivity

LASH was transformed last year into a D-Bus [4] service, effectively making it a seamless part of any Freedesktop.org-compliant environment. At the same time the LASH library's internal protocol mechanism was switched over from TCP to D-Bus.

The conversion to D-Bus of LASH follows in the footsteps of the jackdbus project [5], now a part of the JACK 2 server currently being developed. It naturally follows that LASH now uses D-Bus to communicate with JACK instead of linking against libjack. This serves among other purposes that of making the LASH daemon less vulnerable to bugs in JACK.

D-Bus connectivity introduces many interesting options for those planning to utilise LASH in their programs or desktop activities. These include, but are not limited to:

- Faster and more reliable automatic launching of the LASH service,
- the ability to control LASH via D-Bus browser applications, and
- the possibility to create LASH front-ends or even reimplement the LASH client library itself using any language for which D-Bus bindings exist.

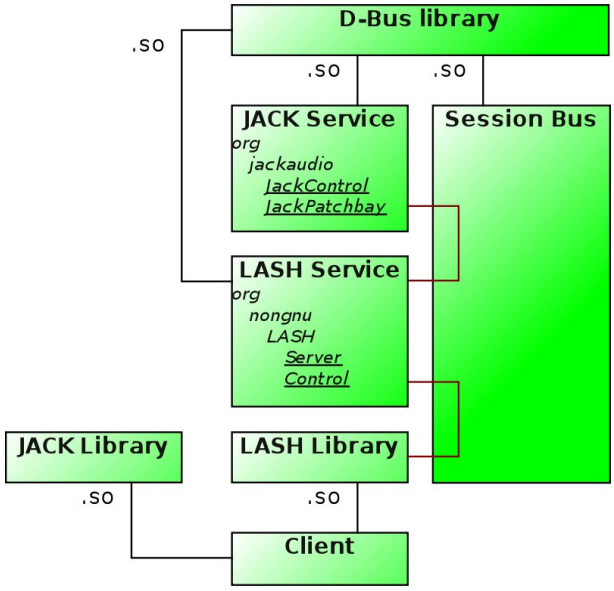


*Diagram 1: How LASH and JACK operate within a D-Bus-enabled desktop setting. A client application, which can be an audio application, a LASH front-end, or a combination of both, links against liblash (and libjack if it is required). The JACK and LASH service daemons link against libdbus to gain desktop messaging abilities. The LASH daemon communicates with both the LASH library and with the JACK daemon using D-Bus messages and signals, which are in turn delivered by the session bus daemon. Shown within the LASH and JACK boxes are the relevant D-Bus interface paths.*

### 4.3 A New API

The existing event-based client API will eventually be replaced with a more flexible callback-based one. An unfinished but working version of the API already exists in the current development branch.

The principal motivation for this change has been the alleged poor design of the old API, but there are some useful and long wanted new features for LASH clients and front-ends as well.

A compatibility layer for the old API is still included with LASH and is currently built by default.

#### 4.3.1 Highlights of the New API

The most obvious change with respect to the old API is the callback model. This change alone brings LASH a step closer to JACK's already familiar programming model. Moreover the new

LASH API has many functions that are named and used similarily to JACK's.

A client registers itself with LASH using lash_client_open(), and callbacks for various purposes are set using the lash_set_*_callback() functions.

Programmers wishing to create LASH front-ends will find it a straightforward process. A front-end will initiate its operation by calling lash_client_open_controller() and setting the required callback with lash_set_control_callback(). There are a handful of functions available for sending control commands to the LASH service.

As a new feature, programs can register themselves as both LASH clients and front-ends at the same time. This is achieved by including the LASH_Server_Interface flag in the parameters to lash_client_open().

In addition to becoming more like JACK's, the new API will do away with some requirements of the old one, such as having to submit the client program's command-line arguments to lash_extract_args(). It will also become unnecessary to separately inform LASH about some previously required details, such as the client's JACK client name.

### 4.4 Deterministic Launching of Clients

As an extra work for Summercode, a mechanism was implemented in LASH for projects to specify client launching order. Future LASH releases will thus be able to launch clients in a completely deterministic order, configurable in the project settings.

### 4.5 Code Cleanups

The LASH codebase has been subjected to the unforgiving (and occasionally destructive) hands of the author of this paper for some time now. This has resulted in fixes to sloppy memory allocations, optimized algorithms, and the removal of assorted redundancies and crud.

The codebase is also slowly approaching a state where things mostly look uniformly designed. (This might have already been the case before Summercode 2008, but it certainly wasn't immediately afterwards.)

### 4.6 Architectural Changes

Today's LASH development code is modular in a much more consistent way than before. It also has a rather nice internal logging framework, a safe memory handling framework, and other components useful in protecting one's sanity while coding in C.

## 5 Current Status

At the time of writing the most recent stable LASH version is 0.5.4. There have been a couple of “pre-release versions” since, but the bugs are still far too many for any serious use.

The current LASH team comprises Dave Robillard, Nedko Arnaudov, and Juuso Alasuutari.

## 6 The Road Ahead

The objective of the LASH project is for LASH support to have become a standard part of any pro-audio application's feature list by the time LASH reaches version 1.0. This calls for not only concentrated effort on behalf of the LASH team, but also continuing feedback and support from external parties.

The feature list for LASH 1.0 is still a moving target. Some of the following features are already partly realized, but all are in need of some kind of attention.

### 6.1 Template Projects

LASH 1.0 will have support for saving projects as templates, and then using them as starting points for newly created projects.

### 6.2 The Project Stack

Although the details are yet undecided, support is planned for flexible management of more than one project at a time. This will mean that a desktop's state may consist of projects that are layered on top of eachother or otherwise side by side in an active state.

### 6.3 Studio Projects

Studio projects are in a way an extension of the project stack idea. They are projects whose sole purpose is to reside between the other projects and the system output ports.

For instance, a studio project may consist of an equalizer program whose parameters are adjusted according to studio acoustics. It then follows that LASH can be used not only for managing transferable projects, but for adjusting the local audio environment as well.

### 6.4 Support for Managing LASH-Unaware Clients

LASH projects will eventually be able to include audio applications that do not themselves communicate with LASH.

Some restrictions apply to how these clients can be controlled; for instance they cannot be commanded to save their state. Instead, this feature

will be useful for including very simple clients for which anything more complex would be overkill.

### 6.5 Networking Support

Support for managing audio sessions over the network is planned for LASH 1.0. The objective is to extend LASH's usefulness in the same way that NetJack extends the usefulness of JACK.

### 6.6 Making LASH Aware of Available Clients

The current development code has preliminary support for keeping a list of available audio applications. This will eventually make it possible for users to browse, launch, and re-launch audio applications from a LASH front-end.

LASH gathers its application list by parsing standard desktop files, making it trivial for clients to add support for "LASH visibility".

### 6.7 Other Important Goals

Other noteworthy items on the LASH 1.0 checklist include:

- Finalizing the new API,
- adding support for importing/exporting projects,
- adding support for undo/redo functionality,
- adding a data storage option for front-ends,
- adding support for external storage locations (such as the memory of a hardware synthesizer),
- improving existing front-ends and tools and creating new ones,
- implementing bugfixes and optimizations, and
- improving code portability and possibly adding C99 compliance.

## 7 Conclusion

LASH has the potential to become a central and enabling part of Linux audio. Development is ongoing, and the LASH team is always happy to accept contributions.

## 8 Acknowledgements

The author of this paper would like to thank LASH's original author, Robert Ham, for obvious reasons, as well as Dave Robillard and Nedko Arnaudov for contributing significant time and effort to developing not only LASH, but many other important Linux audio projects as well.

The LASH team would like to thank all Linux audio software users and developers for their continuing support, as well as the Finnish Centre for Open Source Solutions for sponsoring the development of the LASH with Summercode 2008 and also for sponsoring this presentation for LAC 2009.

## 9 References

[1] http://lash-audio.org/
[2] http://jackaudio.org/
[3] http://www.coss.fi/en/summercode/2008
[4] http://dbus.freedesktop.org/
[5] http://nedko.arnaudov.name/wiki/moin.cgi/LADI#head-19659c1e9cdcebadccc6021f289942ae94a82632