Modular Wayland compositors with wlroots

Drew DeVault et al [*]

December 28, 2017

Abstract

wlroots is a flexible and modular library upon which Wayland compositors can be built. It is designed to maximize several factors: code reusability, interoperability between Wayland compositors, and flexibility for use in novel compositor designs.

[*]Complete list available at `https://github.com/swaywm/wlroots/graphs/contributors`

Contents

1 Background

Wayland is a protocol used to communicate between user applications and a *compositor*. The compositor has ownership over input and output resources– typically displays connected over DVI, HDMI, etc, and keyboards, mice, etc.

wlroots (stylized in lowercase, or as "wlr") is a subproject of the Sway compositor. Originally, Sway was written on top of a similar library called wlc. This library offered abstractions on top of DRM, GBM, GLES, libinput, and so on– allowing Sway to focus on building the software specific to the compositor. However, with time it became clear that wlc's design is too limiting, and many of Sway's long-term goals were difficult to achieve with it.

As a result, we built wlroots. We believe it offers a substantially better basis for development of Wayland compositors than any competing approach. We offer a powerful library that removes most of the heavy lifting from building a Wayland compositor while providing compositor authors as much flexibility in their design as they might gain by doing it all themselves– I've jokingly referred to wlroots as "about 30,000 lines of code you were going to write anyway." We hope that compositors based on wlroots will be written faster, work better, and will be better citizens of the broader Wayland ecosystem.

1.1 Alternatives

Several alternative approaches to building compositors exist.

libweston is the most notable alternative. It is a subproject of the Weston compositor, the reference Wayland compositor. However, it is generally poorly suited to novel compositors that don't use the traditional desktop paradigm. It is also less modular and difficult to break into several pieces. Additionally, as a consequence of being refactored out of an existing compositor, it inherits many design decisions of Weston itself, which may not be amenable to other compositors.

swc and **wlc** are similar. They both strike a reasonable balance between size and features, but both make decisions for the compositor that lock them into particular designs and make novel compositors more difficult to write. The advantage of their approach is that it is easy to use them to build a simple working compositor - in a matter of minutes, not days or weeks.

Another approach is to simply write your compositor entirely in-house. This is the approach favored by **mutter** and **kwin** of the GNOME and KDE projects respectively. The advantage of this approach is the flexibility to design your compositor in any way you please. However, lots of work is duplicated between projects with this approach, which can (and does) lead to compositors which are subtly incompatible with each other.

The approach taken by **wlroots** offers reusability and consistency between compositors based on it, and offloads lots of work from the compositor implementation. This is accomplished without making design decisions on behalf of the compositor implementation: we believe virtually any novel kind of compositor can be built on top of wlroots.

The main disadvantage of our approach is that batteries are not included. More work is required to get a working compositor compared to libweston or wlc. However, you will not hit the limitations of these libraries, and will more quickly arrive at a diverse featureset than with the kwin or mutter approaches.

2 High-level Design

Let's summarize the important high-level components of wlroots.

2.1 Backends

The wlroots backends are similar to backends in other compositors, but are more easily used standalone or in composite. A compositor based on wlroots can use zero or more backends at a time, dynamically adding and removing different backends (and different types of backends) at runtime. Each backend is responsible for providing zero or more input and output devices, as well as an EGL context (and support for maneuvering between EGL contexts is provided). This design enables us to support use cases like multiple GPUs, scripting inputs on your desktop by adding a headless backend, temporarily adding an RDP backend with mirrored outputs to your session, and so on.

We currently offer DRM, libinput, Wayland, X11, and headless backends. We have also written a virtual "multi" backend which acts as a container for multiple backends, with which you may add or remove backends at runtime. Use of this multi-backend is necessary in most non-trivial compositors, for example, to use DRM and libinput at the same time. Also provided is an interface with which compositor authors can create their own backend implementations for more novel systems, and we plan to write an RDP and fbdev backends in the future.

2.2 Shells

Several implementations of Wayland shell state machines are included in wlroots. Their purpose is to provide largely non-opinionated implementations of the various shells available on a Wayland desktop, and to delegate to the compositor for when opinions are necessary. We do not provide a grand unified shell abstraction. Rather, we expose the subtleties of each shell to the compositor author so they may have finer control over how these subtleties are addressed given the constraints of their particular compositor design.

Provided are implementations of wl_shell, xdg_shell, and Xwayland. The latter is quite thorough, on the assumption that the last thing the author of a Wayland compositor wants to do is write an X11 window manager too. However, it is as flexible as any other shell, and you are able to reach in and tweak it to suit your compositor's needs.

2.3 Wayland Protocols

In addition to shell protocols, implementations of various other protocols are provided. This includes core protocols like wl_data_device_manager, wl_compositor, wl_surface, and so on, but also includes some third-party protocols such as Orbital's screenshooter protocol and GTK's primary selection protocol. When appropriate, integrations with wlroots implementations of other protocols may be necessary, such as interactions between wlr_seat and wlr_data_device_manager, or syncronizing GTK's Wayland primary selection with Xwayland.

Each of these protocols, like the shells and everything else, is opt-in. The compositor author chooses if our implementations suit their needs, and if not, their own implementation is easily used instead and interops correctly with the rest of wlroots. For some important protocols (such as wlr_seat), disentangling it from the rest of wlroots can be more difficult– but in these cases we've made our implementation extremely flexible to make it less likely that it will not suit your needs.

2.4 Convenience Utilities

We also provide optional implementations of functionality common to most Wayland compositors, which can be integrated with the rest of wlroots to any degree the compositor author chooses. This includes modules which help you manage the layout of outputs in physical space (and arranging windows on a virtual desktop), help you map input from pointers, touch, etc onto cursors shown on-screen, or to read cursor bitmaps from Xcursor themes for use when your compositor overrides client cursors.

These interfaces are also quite modular and composable, and will generally work in the absence of other wlroots features, or offer more functionality in their presence. The wlr_cursor module, for example, will allow you to limit cursor movement to within the bounds of a wlr_output_layout, should you choose to utilize that as well. We also provide a renderer module, which utilizes gles2 to provides simple rendering support which should be suitable for any compositor that does not require complex shaders or projections.

2.5 Rootston

Bringing all of this together is our reference compositor, Rootston. As Weston is to libweston, Rootston is to wlroots. However, Rootston is more conservative– it is a very simple stacking window manager that has only a few convenience features on top of demonstrating the capabilities of wlroots. This makes it quite unsuitable as a practical end-user compositor. However, in its role of demonstrating the capabilities of wlroots, it does a fine job, and any compositor author hoping to use wlroots will likely find themselves reading, and perhaps copying, large swaths of Rootston's code.

Like the wlroots library itself, Rootston is MIT licensed and users are encouraged to copy and adapt any code they find useful for their own compositors. Many of the batteries that are ”not included” with wlroots can be found in Rootston.

3 Promoting an Interoperable Ecosystem

We believe that it is tremendously important for more Wayland compositors to be extensible by their users and for these tools to work across the entire Wayland desktop ecosystem. No one will switch from X11 to Wayland for security alone– we must make compositors that offer compelling features and do not ask the user to sacrifice the workflows they've grown used to, insecure as they may have been on X11.

We are committed to implementing the necessary Wayland protocol extensions to support these features. Where extensions already exist, we are implementing them: we provide implementations of KDE's server decoration protocol, GTK's primary selection protocol, and Orbital's screenshot protocol; and in the future we intend to implement more. Where protocols do not exist, we are creating new ones.

Consider an on-screen keyboard– getting it right is complicated. There are many keyboard layouts to consider, and localization may be difficult. Support for typing in pinyin (Chinese) or romaji (Japanese) will be difficult to do correctly. Useful features like providing prompts for autocomplete will also take significant developer investment. We could do this many times for many Wayland compositors, or we could push for a single implementation that is compatible with all of our software.

The state of interopability in the Wayland ecosystem is something the wlroots team is deeply concerned about, and we worry that many compositors are not taking this problem seriously. This is why we are implementing the protocols of other desktops and toolkits, and building our own when these prove insufficient. The first of our new protocols is surface layers. We believe this protocol can support numerous use-cases in a single simple protocol, such as:

- desktop shells
- lock screens
- on screen keyboards
- notification daemons
- custom program launchers
- magnifying glass tools

The details of the protocol or its implementation are out of scope for this document, and in any case are changing as we develop it. Here it serves as a call to action for other compositors to participate in its development and implementation, and as an example of the sorts of functionality we're committed to providing on Wayland.

3.1 Security Considerations

A major concern in implementing many of these features is doing them in a secure way. Making these interfaces available to third-party software securely is the subject of ongoing research and discussion, but we do not believe that the issue is too poorly understood to proceed with implementing these features. In wlroots, we have chosen to leave securing these protocols as a matter for the compositor to support via wl_display_set_global_filter.

Let's summarize ongoing efforts by compositors in this area. The Sway compositor has, as of writing, invested the most effort in making privledged protocols available to third parties. This is not without consequences– Sway's current implementation has known flaws. We are comfortable being more liberal in our exploration in this area, hoping that we can become more secure over time. Our justification for this approach is that, at worst, we begin with a system no more secure than X11, but over time we will be able to secure it more as we experiment with different approaches.

Sway's approach today is configuration files, owned by root, which contain a list of paths to executables and the permissions granted to each. To enforce these, we look them up via /proc/[pid]/exe. This has a number of drawbacks we are working to address. First, this is not very effective for interpreted programs, e.g. /usr/bin/python3. Additionally, it may be possible on some platforms and in some conditions to manipulate the value of /proc/[pid]/exe. Utilizing procfs is also not portable.

In the future we're exploring the possibility of only permitting secure access for clients which were forked from Sway. This approach should solve most of the outstanding problems, but other deficiencies remain. Granting permissions to previously un-approved clients at runtime (with user content) requires additional work (something the Orbital compositor has explored). Offering similar permissions to children of those processes when appropriate is also a non-trivial problem.

Collaboration with the broader Wayland community in the matter of security is highly sought after. We ask you to participate in our discussions on the subject; contact information is available at the end of this paper.

4 Project Status

Integrations with wlroots are well underway in the Wayland community.

Sway is a Wayland compositor compatible with the i3 window manager and is the motivating force behind wlroots, and our port from wlc to wlroots is going well. We've found it most effective to set aside our current code and start from scratch, importing large parts of the old code as it becomes necessary. Porting is no simple process, but it seems that it will be well worth it. Dozens of features that our users have been asking for have been blocked behind this migration for a long time, and many are already supported in our port[1].

Way-Cooler is also working on porting their compositor from wlc to wlroots. In order to do so, they are developing **wlroots-rs**, a Rust wrapper over the wlroots API. They intend to utilize this to implement a large subset of the Awesome window manager's Lua API, providing a compatible alternative to AwesomeWM in a similar fashion as Sway does for i3. Their first order of business is providing an equivalent of Rootston written in Rust.

waymonad is a new compositor based on wlroots– not a port of another compositor. Like Sway fills i3's niche and Way-Cooler fills AwesomeWM's niche, waymonad intends to fill Xmonad's niche (though it seems that perfect compatibility is not presently a concern). To this end **hsroots** is underway, Haskell wrappings for the wlroots API.

We've heard murmurings of a compositor written in Go as well. We are excited about and prepared to support any other compositors that want to port their plumbing to wlroots, as well as any new compositors that would like to use wlroots from the outset. As the Sway port and other projects make progress, we continue to prove the effecacy of the wlroots approach and stabilize its API, and we are pushing towards 1.0 soon. Please reach out if you have feedback or wish to participate.

`https://github.com/swaywm/wlroots`

`irc://irc.freenode.net/#sway-devel`

[1]Enumerated here: `https://github.com/swaywm/sway/issues/1524`