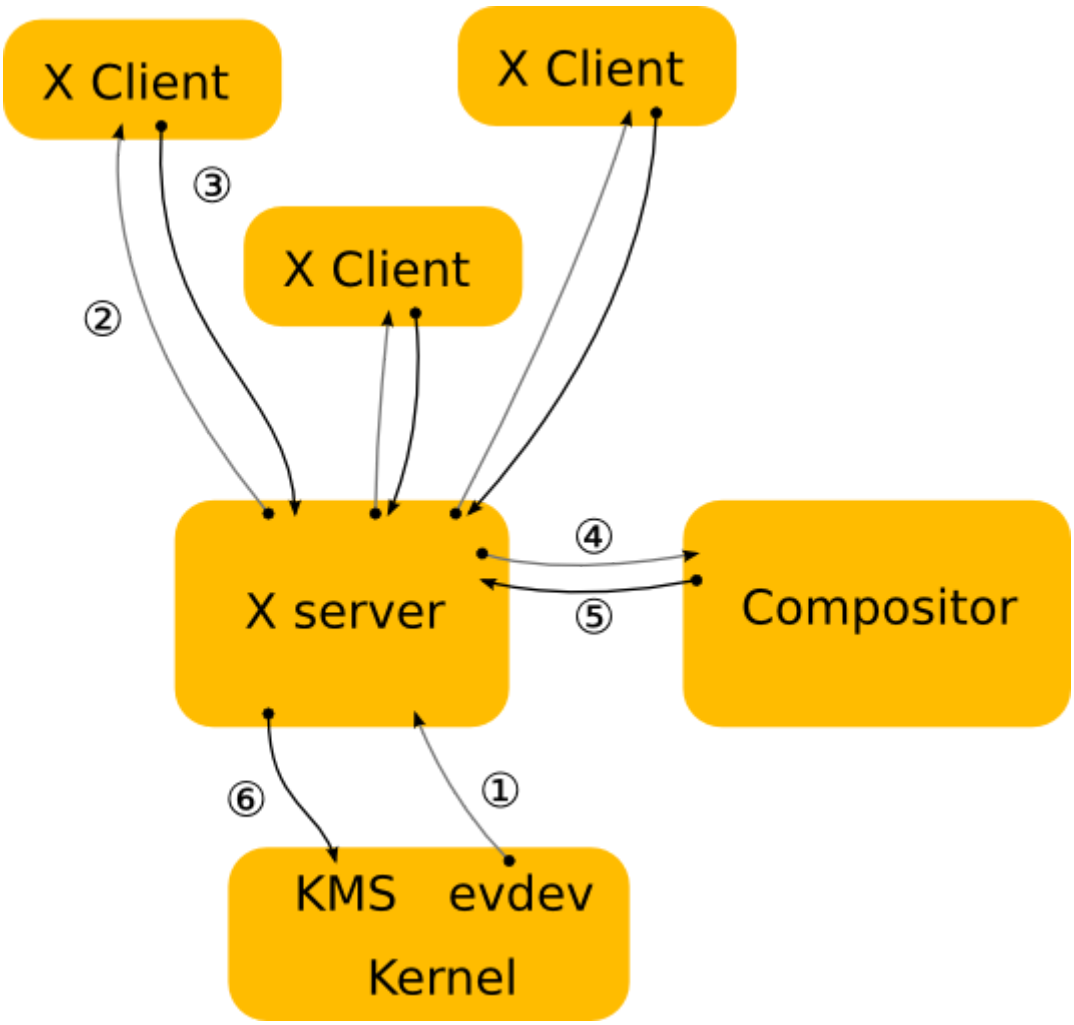
Wayland Architecture

A good way to understand the wayland architecture and how it is different from X is to follow an event from the input device to the point where the change it affects appears on screen.

This is where we are now with X:

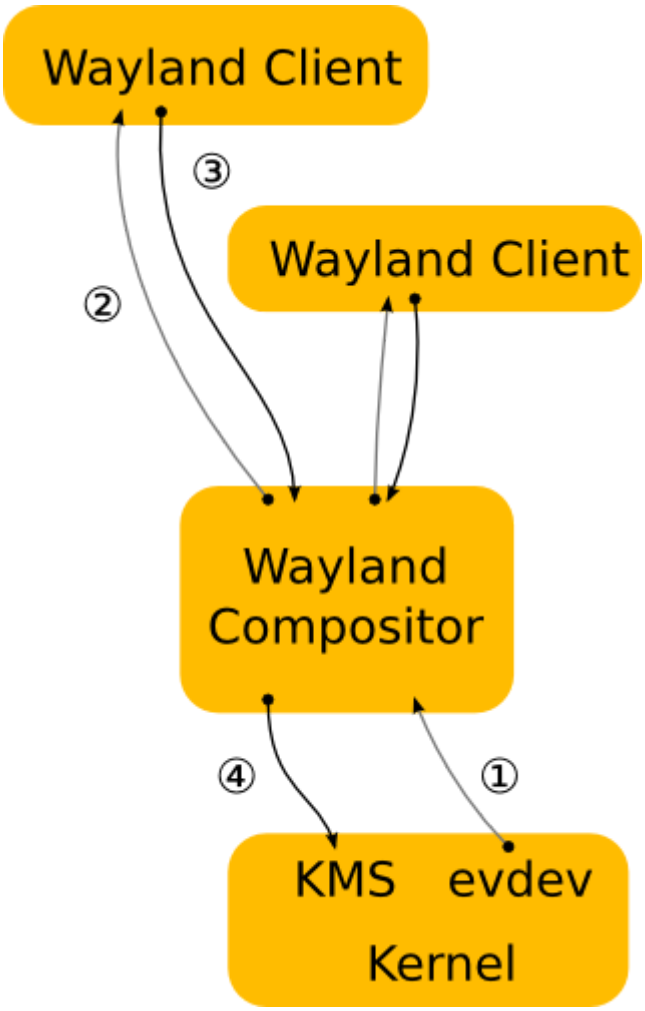


1. The kernel gets an event from an input device and sends it to X through the evdev input driver. The kernel does all the hard work here by driving the device and translating the different device specific event protocols to the linux evdev input event standard.
2. The X server determines which window the event affects and sends it to the clients that have selected for the event in question on that window. The X server doesn't actually know how to do this right, since the window location on screen is controlled by the compositor and may be transformed in a number of ways that the X server doesn't understand (scaled down, rotated, wobbling, etc).
3. The client looks at the event and decides what to do. Often the UI will have to change in response to the event - perhaps a check box was clicked or the pointer entered a button that must be highlighted. Thus the client sends a rendering request back to the X server.
4. When the X server receives the rendering request, it sends it to the driver to let it program the hardware to do the rendering. The X server also calculates the bounding region of the rendering, and sends that to the compositor as a *damage event*.
5. The damage event tells the compositor that something changed in the window and that it has to recomposite the part of the screen where that window is visible. The compositor is responsible for rendering the entire screen contents based on its scenegraph and the contents of the X windows. Yet, it has to go through the X server to render this.
6. The X server receives the rendering requests from the compositor and either copies the compositor back buffer to the front buffer or does a pageflip. In the general case, the X server has to do this step so it can account for overlapping windows, which may require clipping and determine whether or not it can page flip. However, for a compositor, which is always fullscreen, this is another unnecessary context switch.

As suggested above, there are a few problems with this approach. The X server doesn't have the information to decide which window should receive the event, nor can it transform the screen coordinates to window-local coordinates. And even though X has handed responsibility for the final painting of the screen to the compositing manager, X still controls the front buffer and modesetting. Most of the complexity that the X server used to handle is now available in the kernel or self contained libraries (KMS, evdev, mesa, fontconfig, freetype, cairo, Qt, etc). In general, the X server is now just a middle man that introduces an extra step between applications and the compositor and an extra step between the compositor and the hardware.

In wayland the compositor *is* the display server. We transfer the control of KMS and evdev to the compositor. The wayland

protocol lets the compositor send the input events directly to the clients and lets the client send the damage event directly to the compositor:



1. The kernel gets an event and sends it to the compositor. This is similar to the X case, which is great, since we get to reuse all the input drivers in the kernel.
2. The compositor looks through its scenegraph to determine which window should receive the event. The scenegraph corresponds to what's on screen and the compositor understands the transformations that it may have applied to the elements in the scenegraph. Thus, the compositor can pick the right window and transform the screen coordinates to window-local coordinates, by applying the inverse transformations. The types of transformation that can be applied to a window is only restricted to what the compositor can do, as long as it can compute the inverse transformation for the input events.
3. As in the X case, when the client receives the event, it updates the UI in response. But in the wayland case, the rendering happens in the client, and the client just sends a request to the compositor to indicate the region that was updated.
4. The compositor collects damage requests from its clients and then recomposites the screen. The

compositor can then directly issue an ioctl to schedule a pageflip with KMS.

Wayland Rendering

One of the details I left out in the above overview is how clients actually render under wayland. By removing the X server from the picture we also removed the mechanism by which X clients typically render. But there's another mechanism that we're already using with DRI2 under X: *direct rendering*. With direct rendering, the client and the server share a video memory buffer. The client links to a rendering library such as OpenGL that knows how to program the hardware and renders directly into the buffer. The compositor in turn can take the buffer and use it as a texture when it composites the desktop. After the initial setup, the client only needs to tell the compositor which buffer to use and when and where it has rendered new content into it.

This leaves an application with two ways to update its window contents:

1. Render the new content into a new buffer and tell the compositor to use that instead of the old buffer. The application can allocate a new buffer every time it needs to update the window contents or it can keep two (or more) buffers around and cycle between them. The buffer management is entirely under application control.
2. Render the new content into the buffer that it previously told the compositor to use. While it's possible to just render directly into the buffer shared with the compositor, this might race with the compositor. What can happen is that repainting the window contents could be interrupted by the compositor repainting the desktop. If the application gets interrupted just after clearing the window but before rendering the contents, the compositor will texture from a blank buffer. The result is that the application window will flicker between a blank window or half-rendered content. The traditional way to avoid this is to render the new content into a back buffer and then copy from there into the compositor surface. The back buffer can be allocated on the fly and just big enough to hold the new content, or the application can keep a buffer around. Again, this is under application control.

In either case, the application must tell the compositor which area of the surface holds new contents. When the application renders directly to the shared buffer, the compositor needs to be noticed that there is new content. But also when exchanging buffers, the compositor doesn't assume anything changed, and needs a request from the application before it will repaint the desktop. The idea that even if an application passes a new buffer to the compositor, only a small part of the buffer may be different, like a blinking cursor or a spinner.

Hardware Enabling for Wayland

Typically, hardware enabling includes modesetting/display and EGL/GLES2. On top of that, Wayland needs a way to share buffers efficiently between processes. There are two sides to that, the client side and the server side.

On the client side we've defined a Wayland EGL platform. In the EGL model, that consists of the native types (EGLNativeDisplayType, EGLNativeWindowType and EGLNativePixmapType) and a way to create those types. In other words, it's the glue code that binds the EGL stack and its buffer sharing mechanism to the generic Wayland API. The EGL stack is expected to provide an implementation of the Wayland EGL platform. The full API is in the wayland-egl.h header. The open source implementation in the mesa EGL stack is in platform_wayland.c.

Under the hood, the EGL stack is expected to define a vendor-specific protocol extension that lets the client side EGL stack communicate buffer details with the compositor in order to share buffers. The point of the wayland-egl.h API is to abstract that away and just let the client create an EGLSurface for a Wayland surface and start rendering. The open source stack uses the drm Wayland extension, which lets the client discover the drm device to use and authenticate and then share drm (GEM) buffers with the compositor.

The server side of Wayland is the compositor and core UX for the vertical, typically integrating task switcher, app launcher, lock screen in one monolithic application. The server runs on top of a modesetting API (kernel modesetting, OpenWF Display or similar) and composites the final UI using a mix of EGL/GLES2 compositor and hardware overlays if available. Enabling modesetting, EGL/GLES2 and overlays is something that should be part of standard hardware bringup. The extra requirement for Wayland enabling is the EGL_WL_bind_wayland_display extension that lets the compositor create an EGLImage from a generic Wayland shared buffer. It's similar to the EGL_KHR_image_pixmap extension to create an EGLImage from an X pixmap.

The extension has a setup step where you have to bind the EGL display to a Wayland display. Then as the compositor receives generic Wayland buffers from the clients (typically when the client calls eglSwapBuffers), it will be able to pass the struct wl_buffer pointer to eglCreateImageKHR as the EGLClientBuffer argument and with EGL_WAYLAND_BUFFER_WL as the target. This will create an EGLImage, which can then be used by the compositor as a texture or passed to the modesetting code to use as an overlay plane. Again, this is implemented by the vendor specific protocol extension, which on the server side will receive the driver specific details about the shared buffer and turn that into an EGL image when the user calls eglCreateImageKHR.