The Wayland Display Server

Kristian Hgsberg
krh@bitplanet.net

July 26, 2010

1 Wayland Overview

- wayland is a protocol for a new display server.
- wayland is an implementation

1.1 Replacing X11

Over time, a lot of functionality have slowly moved out of the X server and into client-side libraries or kernel drivers. One of the first components to move out was font rendering, with freetype and fontconfig providing an alternative to the core X fonts. Direct rendering OpenGL as a graphics driver in a client side library. Then cairo came along and provided a modern 2D rendering library independent of X and compositing managers took over control of the rendering of the desktop. Recently with GEM and KMS in the Linux kernel, we can do modesetting outside X and schedule several direct rendering clients. The end result is a highly modular graphics stack.

1.2 Make the compositing manager the display server

Wayland is a new display server building on top of all those components. We are trying to distill out the functionality in the X server that is still used by the modern Linux desktop. This turns out to be not a whole lot. Applications can allocate their own off-screen buffers and render their window contents by themselves. In the end, whats needed is a way to present the resulting window surface to a compositor and a way to receive input. This is what Wayland provides, by piecing together the components already in the eco-system in a slightly different way.

X will always be relevant, in the same way Fortran compilers and VRML browsers are, but its time that we think about moving it out of the critical path and provide it as an optional component for legacy applications.

2 Wayland protocol

2.1 Basic Principles

The wayland protocol is an asynchronous object oriented protocol. All requests are method invocations on some object. The request include an object id that uniquely identifies an object on the server. Each object implements an interface and the requests include an opcode that identifies which method in the interface to invoke.

The wire protocol is determined from the C prototypes of the requests and events. There is a straight forward mapping from the C types to packing the bytes in the request written to the socket. It is possible to map the events and requests to function calls in other languages, but that hasn't been done at this point.

The server sends back events to the client, each event is emitted from an object. Events can be error conditions. The event includes the object id and the event opcode, from which the client can determine the type of event. Events are generated both in repsonse to a request (in which case the request and the event constitutes a round trip) or spontanously when the server state changes.

- state is broadcast on connect, events sent out when state change. client must listen for these changes and cache the state. no need (or mechanism) to query server state.
- server will broadcast presence of a number of global objects, which in turn will broadcast their current state

2.2 Connect Time

- no fixed format connect block, the server emits a bunch of events at connect time
- presence events for global objects: output, compositor, input devices

2.3 Security and Authentication

- mostly about access to underlying buffers, need new drm auth mechanism (the grant-to ioctl idea), need to check the cmd stream?
- getting the server socket depends on the compositor type, could be a system wide name, through fd passing on the session dbus. or the client is forked by the compositor and the fd is already opened.

2.4 Creating Objects

- client allocates object ID, uses range protocol
- server tracks how many IDs are left in current range, sends new range when client is about to run out.

2.5 Compositor

The compositor is a global object, advertised at connect time.

Interface `compositor`
Requests
`create_surface(id)`
`commit()`
Events
`device(device)`
`acknowledge(key, frame)`
`frame(frame, time)`

- a global object
- broadcasts drm file name, or at least a string like drm:/dev/card0
- commit/ack/frame protocol

2.6 Surface

Created by the client.

Interface `surface`
Requests
`destroy()`
`attach()`
`map()`
`damage()`
Events
no events

Needs a way to set input region, opaque region.

2.7 Input

Represents a group of input devices, including mice, keyboards. Has a keyboard and pointer focus. Global object. Pointer events are delivered in both screen coordinates and surface local coordinates.

Interface `cache`
Requests
no requests
Events
`motion(x, y, sx, sy)`
`button(button, state, x, y, sx, sy)`
`key(key, state)`
`pointer_focus(surface)`
`keyboard_focus(surface, keys)`

Talk about:

- keyboard map, change events
- xkb on wayland
- multi pointer wayland

A surface can change the pointer image when the surface is the pointer focus of the input device. Wayland doesn't automatically change the pointer image when a pointer enters a surface, but expects the application to set the cursor it wants in response the the motion event. The rationale is that a client has to manage changing pointer images for UI elements within the surface in response to motion events anyway, so we'll make that the only mechanism for setting changing the pointer image. If the server receives a request to set the pointer image after the surface loses pointer focus, the request is ignored. To the client this will look like it successfully set the pointer image.

The compositor will revert the pointer image back to a default image when no surface has the pointer focus for that device. Clients can revert the pointer image back to the default image by setting a NULL image.

What if the pointer moves from one window which has set a special pointer image to a surface that doesn't set an image in response to the motion event? The new surface will be stuck with the special pointer image. We can't just revert the pointer image on leaving a surface, since if we immediately enter a surface that sets a different image, the image will flicker. Broken app, I suppose.

2.8 Output

A output is a global object, advertised at connect time or as they come and go.

Interface `output`
Requests
no requests
Events
`geometry(width, height)`

- laid out in a big (compositor) coordinate system
- basically xrandr over wayland
- geometry needs position in compositor coordinate system
- events to advertise available modes, requests to move and change modes

2.9 Shared object cache

Cache for sharing glyphs, icons, cursors across clients. Lets clients share identical objects. The cache is a global object, advertised at connect time.

Interface `cache`
Requests
`upload(key, visual, bo, stride, width, height)`
Events

`item(key, bo, x, y, stride)`
`retire(bo)`

- Upload by passing a visual, bo, stride, width, height to the cache.
- Upload returns a bo name, stride, and x, y location of object in the buffer. Clients take a reference on the atlas bo.
- Shared objects are refcounted, freed by client (when purging glyphs from the local cache) or when a client exits.
- Server can't delete individual items from an atlas, but it can throw out an entire atlas bo if it becomes too sparse. The server sends out an `retire` event when this happens, and clients must throw away any objects from that bo and reupload. Between the server dropping the atlas and the client receiving the retire event, clients can still legally use the old atlas since they have a ref on the bo.
- cairo needs to hook into the glyph cache, and maybe also a way to create a read-only surface based on an object form the cache (icons).

  `cairo_wayland_create_cached_surface(surface-data).`

2.10 Drag and Drop

Multi-device aware. Orthogonal to rest of wayland, as it is its own toplevel object. Since the compositor determines the drag target, it works with transformed surfaces (dragging to a scaled down window in expose mode, for example).

Issues:

- we can set the cursor image to the current cursor + dragged object, which will last as long as the drag, but maybe an request to attach an image to the cursor will be more convenient?
- Should drag.send() destroy the object? There's nothing to do after the data has been transferred.
- How do we marshall several mime-types? We could make the drag setup a multi-step operation: dnd.create, drag.offer(mime-type1, drag.offer(mime-type2), drag.activate(). The drag object could send multiple offer events on each motion event. Or we could just implement an array type, but that's a pain to work with.
- Middle-click drag to pop up menu? Ctrl/Shift/Alt drag?

- Send a file descriptor over the protocol to let initiator and source exchange data out of band?
- Action? Specify action when creating the drag object? Ask action?

New objects, requests and events:

- New toplevel dnd global. One method, creates a drag object: `dnd.start(new object id, surface, input device, mime types).` Starts drag for the device, if it's grabbed by the surface. drag ends when button is released. Caller is responsible for destroying the drag object.
- Drag object methods:

  `drag.destroy(id)`, destroy drag object.

  `drag.send(id, data)`, send drag data.

  `drag.accept(id, mime type)`, accept drag offer, called by target surface.
- Drag object events:

  `drag.offer(id, mime-types)`, sent to potential destination surfaces to offer drag data. If the device leaves the window or the originator cancels the drag, this event is sent with mime-types = NULL.

  `drag.target(id, mime-type)`, sent to drag originator when a target surface has accepted the offer. if a previous target goes away, this event is sent with mime-type = NULL.

  `drag.data(id, data)`, sent to target, contains dragged data. ends transaction on the target side.

Sequence of events:

- The initiator surface receives a click (which grabs the input device to that surface) and then enough motion to decide that a drag is starting. Wayland has no subwindows, so it's entirely up to the application to decide whether or not a draggable object within the surface was clicked.
- The initiator creates a drag object by calling the `create_drag` method on the dnd global object. As for any client created object, the client allocates the id. The `create_drag` method also takes the originating surface, the device that's dragging and the mime-types supported. If the surface has indeed grabbed the device passed in, the server will create an active drag object for the device. If the grab was released in the meantime, the drag object will be in-active, that is, the same state as when the grab is released. In that case, the client will receive a button up event, which will let it know that the drag finished. To the client it will look like the drag was immediately cancelled by the grab ending.

  The special mime-type application/x-root-target indicates that the initiator is looking for drag events to the root window as well.

- To indicate the object being dragged, the initiator can replace the pointer image with an larger image representing the data being dragged with the cursor image overlaid. The pointer image will remain in place as long as the grab is in effect, since the initiating surface keeps pointer focus, and no other surface receives enter events.

- As long as the grab is active (or until the initiator cancels the drag by destroying the drag object), the drag object will send `offer` events to surfaces it moves across. As for motion events, these events contain the surface local coordinates of the device as well as the list of mime-types offered. When a device leaves a surface, it will send an `offer` event with an empty list of mime-types to indicate that the device left the surface.

- If a surface receives an offer event and decides that it's in an area that can accept a drag event, it should call the `accept` method on the drag object in the event. The surface passes a mime-type in the request, picked from the list in the offer event, to indicate which of the types it wants. At this point, the surface can update the appearance of the drop target to give feedback to the user that the drag has a valid target. If the `offer` event moves to a different drop target (the surface decides the offer coordinates is outside the drop target) or leaves the surface (the offer event has an empty list of mime-types) it should revert the appearance of the drop target to the inactive state. A surface can also decide to retract its drop target (if the drop target disappears or moves, for example), by calling the accept method with a NULL mime-type.

- When a target surface sends an `accept` request, the drag object will send a `target` event to the initiator surface. This tells the initiator that the drag currently has a potential target and which of the offered mime-types the target wants. The initiator can change the pointer image or drag source appearance to reflect this new state. If the target surface retracts its drop target of if the surface disappears, a `target` event with a NULL mime-type will be sent.

  If the initiator listed application/x-root-target as a valid mime-type, dragging into the root window will make the drag object send a `target` event with the application/x-root-target mime-type.

- When the grab is released (indicated by the button release event), if the drag has an active target, the initiator calls the `send` method on the drag object to send the data to be transferred by the drag operation, in the format requested by the target. The initiator can then destroy the drag object by calling the `destroy` method.

- The drop target receives a `data` event from the drag object with the requested data.

MIME is defined in RFC's 2045-2049. A registry of MIME types is maintained by the Internet Assigned Numbers Authority (IANA).

ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/

3 Types of compositors

3.1 System Compositor

- ties in with graphical boot
- hosts different types of session compositors
- lets us switch between multiple sessions (fast user switching, secure/personal desktop switching)
- multiseat
- linux implementation using libudev, egl, kms, evdev, cairo
- for fullscreen clients, the system compositor can reprogram the video scanout address to source fromt the client provided buffer.

3.2 Session Compositor

- nested under the system compositor. nesting is feasible because protocol is async, roundtrip would break nesting
- gnome-shell
- moblin
- compiz?
- kde compositor?
- text mode using vte
- rdp session
- fullscreen X session under wayland
- can run without system compositor, on the hw where it makes sense
- root window less X server, bridging X windows into a wayland session compositor

3.3 Embbedding Compositor

X11 lets clients embed windows from other clients, or lets client copy pixmap contents rendered by another client into their window. This is often used for applets in a panel, browser plugins and similar. Wayland doesn't directly allow this, but clients can communicate GEM buffer names out-of-band, for example, using d-bus or as command line arguments when the panel launches the applet. Another option is to use a nested wayland instance. For this, the wayland server will have to be a library that the host application links to. The host application will then pass the wayland server socket name to the embedded application, and will need to implement the wayland compositor interface. The host application composites the client surfaces as part of it's window, that is, in the web page or in the panel. The benefit of nesting the wayland server is that it provides the requests the embedded client needs to inform the host about buffer updates and a mechanism for forwarding input events from the host application.

- firefox embedding flash by being a special purpose compositor to the plugin

4 Implementation

what's currently implemented

4.1 Wayland Server Library

`libwayland-server.so`

- implements protocol side of a compositor
- minimal, doesn't include any rendering or input device handling
- helpers for running on egl and evdev, and for nested wayland

4.2 Wayland Client Library

`libwayland.so`

- minimal, designed to support integration with real toolkits such as Qt, GTK+ or Clutter.
- doesn't cache state, but lets the toolkits cache server state in native objects (GObject or QObject or whatever).

4.3 Wayland System Compositor

- implementation of the system compositor
- uses libudev, eagle (egl), evdev and drm
- integrates with ConsoleKit, can create new sessions
- allows multi seat setups
- configurable through udev rules and maybe /etc/wayland.d type thing

4.4 X Server Session

- xserver module and driver support
- uses wayland client library
- same X.org server as we normally run, the front buffer is a wayland surface but all accel code, 3d and extensions are there
- when full screen the session compositor will scan out from the X server wayland surface, at which point X is running pretty much as it does natively.