Wayland book; notes

Contents

[1] https://github.com/tobiasjakobi/libdrm
[2] https://mesa3d.org/
[3] https://wayland.freedesktop.org/libinput/doc/latest/
[4] https://en.wikipedia.org/wiki/Udev and https://github.com/eudev-project/eudev
[5] https://xkbcommon.org/
[6] https://www.pixman.org/
[7] https://wayland.freedesktop.org/

1. Introduction

Wayland is the next generation display server for Unix-systems.

1.1. High-level design

Computers have multiple **input** and **output** devices, they are responsible for receiving information from you and displaying information to you.

Output devices are generally displays. These resources are shared between all applications, and the role of the **Wayland compositor** is to dispatch input events to the appropriate **Wayland client** and to display their windows in their appropriate place on your outputs.

The process of bringing together all of your application windows for displaying on an output is called **composing**.

1.2. In practice

Multiple distinct software components are part of the graphics/display stack. Some tools of the tools found part of the Linux desktop stack are:

- Mesa for rendering.
- Linux KMS/DRM.
- Buffer allocation with GBM.
- Userspace libdrm library, libinput, evdev, etc.

1.3. The hardware

Interfacing input and output devices is done by several components inside the operating system. This can be interfaces for USB, PCI, etc. Hardware has little concept of what applications are running on the system. The hardware only provides an interface with which it can be commanded to perform work, and do what it is told, regardless of who tells it so. For this reason, only one component is allowed to talk to hardware, this is the **kernel**.

1.4. The kernel

The job of the kernel is to provide an abstraction over the hardware, so that it can be safely accessed from the **userspace**. The userspace is also where the Wayland compositor runs.

The graphics abstraction in the Linux kernel is called **DRM** or **direct rendering manager**[8]. DRM tasks the GPU with work from the userspace. The displays themselves are configured by a subsystem of DRM known as Linux **KMS** or **kernel mode setting**[9].

Input devices are abstracted through an interface called **evdev**[10].

Most kernel interfaces are exposed to the userspace by the way of special files in `/dev`. In the case of DRM, these files are in `/dev/dri`:

- In the form of a primary node for privileged operations like modesetting.
- In the form of render nodes for unprivileged operations like rendering or video decoding.

For evdev, the device nodes are in `/dev/input/event*`.

1.5. The userspace

Applications in the userspace are isolated from the hardware and must work with it via the device nodes provided by the kernel.

1.5.1. `libdrm`[11]

`libdrm` is the userspace portion of the DRM subsystem. It's a library providing an C API for interfacing with DRM. `libdrm` is used by Wayland compositors to do mode setting and other DRM operations. It is generally not used by the Wayland clients directly.

1.5.2. Mesa[12]

Mesa is one of the core parts of the Linux graphics stack. It provides an abstraction over `libdrm` known as **GBM** (Generic Buffer Management) library for allocating buffers on the GPU. It also provides vendor-optimized implementations of OpenGL, etc.

[8]https://en.wikipedia.org/wiki/Direct_Rendering_Manager
[9]https://www.kernel.org/doc/html/v4.15/gpu/drm-kms.html
[10]https://en.wikipedia.org/wiki/Evdev
[11]https://github.com/tobiasjakobi/libdrm
[12]https://mesa3d.org/

1.5.3. `libinput`[13]

`libinput` is the userspace abstraction library for evdev. It's responsibility is to receive input events from the kernel, decoding them, and passing them to the Wayland compositor. The Wayland compositor requires special permission to use the evdev files, forcing the Wayland clients to go through the compositor to receive input events (for security reasons).

1.5.4. `(e)udev`[14]

Dealing with the appearance of new devices from the kernel, as well as configuring permissions for the resulting device nodes in `/dev`, and sending word of the changes to the applications running on the system, is a responsibility that falls onto the userspace. Most systems use `udev` or `eudev` for this purpose. The Wayland compositor uses udev to interface, enumerate and notify about changes with input devices.

1.5.5. `xkbcommon`[15]

XKB is the original keyboard handling subsystem for Xorg server. Its now an independent keyboard library. Libinput delivers keyboard events in the form of scan codes, which are keyboard dependent. XKB translates the scan codes into generic key "symbols". It also contains a state machine which knows how to process key combinations.

1.5.6. `pixman`[16]

Library for efficient manipulation of pixel buffers.

1.5.7. `libwayland`[17]

`libwayland` handles most of the low-level wire protocol.

1.6. The Wayland Package

The Wayland package consists of `libwayland-client`, `libwayland-server`, `wayland-scanner` and `wayland.xml`. When installed they can be found in `/usr/lib & /usr/include`, `/usr/bin` and `/usr/share/wayland/`. This package is the most popular implementation of Wayland.

- `wayland.xml`: Wayland protocols are defined by the XML files.
- `wayland-scanner`: Processes the XML files and generates code from them[18].
- `libwayland`: There are two libraries, one for the client side of the wire protocol and one for the server side.

2. Protocol Design

The Wayland protocol consists of serveral layers of abstractions:

1. Basic wire protocol format, which is a stream of messages decodable with agreed upon interfaces.
2. Procedures for enumerating interfaces
3. Procedures for creating resources which conform to these interfaces
4. Procedures for exchanging messages about interfaces.

On top of this we also have some broader patters which are frequently used in Wayland protocol design.

2.1. Wire protocol basics

The wire protocol is a stream of 32-bit values, encoded with the host's byte order. The protocol consists of the following primitive types:

- `int, uint`: 32-bit (un)signed integer.
- `fixed`: 24.8 bit signed fixed-point numbers.
- `object`: 32-bit object ID.
- `new_id`: 32-bit object ID which allocates that object when received.

The following other types are also used:

- `string`: It is prefixed with a 32-bit integer specifying its length in bytes, followed by the string contents and a `NUL` terminator, padded to 32 bits with undefined data.
- `array`: A blob of arbitrary data, prefix with a 32-bit integer of its length, then the contents, padded to 32 bits.
- `fd`: 0-bit value on the primary transport, but transfers a file descriptor to the other end using ancillary data in the Unix domain socket message (`msg_control`).

[13]https://wayland.freedesktop.org/libinput/doc/latest/

[14]https://en.wikipedia.org/wiki/Udev and https://github.com/eudev-project/eudev

[15]https://xkbcommon.org/

[16]https://www.pixman.org/

[17]https://wayland.freedesktop.org/

[18]Other scanners also exist, like `wayland-rs` and `waymonad-scanner`

- `enum`: A single value or bitmap from an enumeration of known constants, encoded into a 32-bit integer.

2.1.1. Messages

The wire protocol is a stream of messages built with these primitives. Every message is an event (server to client) or request (client to server) which acts upon an *object*.

Structure of the message

1. **header**: Two words
   - First word is the affected object ID.
   - Second word is two 16-bit values:
     - The upper 16 bits are the size of the message (including the header).
     - The lower 16 bits are the event or request opcode.
2. **arguments**: Based on a agreed upon in advance message signature.
   - The recipient looks up the object ID's interface and the event or request defined by its opcode to determine the signature and nature of the message.

To understand a message, the client and server have to establish the objects in the first place. Object ID `1` is pre-allocated as the Wayland display `singleton`, and can be used to bootstrap other objects.

2.1.2. Object IDs

When a message comes in with a `new_id` argument, the sender allocates an object ID for it. The interface used for this object is established through additional arguments, or agreed upon in advance for that request/event. This object ID can be used in future messages as the first word of the header, or as an `object_id` argument. The client allocates IDs in the range of `[1, 0xFEFFFFFF]`, and the server allocates IDs in the range of `[0xFF000000, 0xFFFFFFFF]`. IDs begin at the lower end of the range.

An object ID of `0` represents a null object; that is, a non-existent object or the explicit lack of an object.

2.1.3. Transports

The Unix domain socket is used for message transportation. Unix sockets are used because of **file descriptor messages**. This is the most practical transport capable of transferring file descriptors between processes, which is necessary for large data transfers such as keymaps, pixel buffers, and mostly clipboard contents. In theory other transports are also possible.

To find the Unix socket to connect to, most implementation do the same as libwayland:

1. If `WAYLAND_SOCKET` is set, interpret is as a file descriptor number on which the connection is already established, assuming that the parent process configured the connection for us.
2. If `WAYLAND_DISPLAY` is set, concat with `XDG_RUNTIME_DIR` to form the path to the Unix socket.
3. Assure the socket name is `wayland-0` and concat with `XDG_RUNTIME_DIR` to form the path to the Unix socket.
4. Give up.

2.2. Interfaces, requests and event

The protocol works by issuing *requests* and *events* that act on *objects*. Each object has an *interface* which defines what requests and events are possible, and the *signature* of each. Let's consider an example interface: `wl_surface`.

2.2.1. Requests

A surface is a box of pixels that can be displayed on-screen. It's one of the primitives, used for building application windows. One of its *requests*, send from the client to the server, is "damage", which is used by the client to indicate that some part of the surface has changed and needs to be redrawn.

2.2.2. Events

Events are sent from the server to the client. One of the events the server can send to the surface is "enter", which it sends when the surface is being displayed on a specific output.

2.2.3. Interfaces

The interfaces which define the list of requests and events, the opcodes associated with each, and the signatures with which you can decode the messages, are agreed upon in advance.

Interfaces are defined through the XML files (`wayland.xml`) mentioned beforehand. Each interface is defined in this file, along with its requests and events, and their respective signatures.

During the XML file processing, we assign each request and event an opcode in the order that they appear, numbered from `0` and incrementing independently. Combined with the list of arguments, you can decode the request or event when it comes in over the wire, and based on the documentation in the XML file you can decide how to program your software to behave accordingly. This usually comes in the form of code generation.

2.3. Protocol design patterns

The following are some key concept used in the design of both the Wayland protocol and the protocol extensions.

2.3.1. Atomicity

Atomicity is the most important design pattern. Most interfaces allow you tu update them transactionally, using several requests to build up a new representations of its state, then committing them all at once.

The interface includes separate requests for configuring each property of an object. These are applied to a *pending* state. When the **commit** is sent, the pending state gets merged into the *current* state. This enables atomic updates within a single frame, resulting in no tearing or partially updates Windows.

2.3.2. Resource lifetimes

We wish to avoid sending events or requests to invalid objects. Interfaces which define resources that have finite lifetimes will often include requests and events through which the client or server can state their intention to no longer send requests or events for that object. Only once both sides agree to this (asynchronously) do they destroy the resources they allocated for that object.

Wayland is a fully asynchronous protocol. Messages are guaranteed to arrive in the order they were sent, but only with the respect to one sender. The client and server need to correctly handle the objects until a confirmation of destruction in received.

3. `libwayland` in depth

`libwayland` is the most popular Wayland implementation. The library includes a few simple primitives and a pre-compiled version of `wayland.xml`, the core Wayland protocol.

3.1. `wayland-util` primitives

`wayland-util.h` defines a number of structures, utility functions and macros for Wayland applications.

Among these are:

- Structures for **marshalling**[19] and unmarshalling Wayland protocol messages in generated code.
- A linked list `wl_list` implementation.
- An array `wl_array` implementation.
- Utilities for conversion between Wayland scalar types and C scalar types.
- Debug logging.

3.2. `wayland-scanner`

`wayland-scanner` is packed with the Wayland package. It is used to generate C headers & glue code from the XML files.

Generally you run `wayland-scanner` at build time, then compile and link your application to the glue code.

3.3. Proxies and resources

An object is an entity known to the client and server that has some state, changes to which are negotiated over the wire.

On the client side, `libwayland` refers to these objects through the `wl_proxy` interface. This is a proxy for the abstract object, and provides functions which are indirectly used by the client to marshall requests. On the server side, objects are abstracted through `wl_resource`, which is similar to the proxy but also tracks which object belongs to which client.

3.4. Interfaces and listeners

Interfaces and listeners are the highest abstraction in `libwayland`, primitives, proxies and listeners are just low level implementations specific to each `interface` and `listener` generated by `wayland-scanner`.

Both the client and server listen for messages using `wl_listener`. The server-side code for interfaces and listeners is identical, but reversed. When a message is received, it first looks up the object ID and its interface, then uses that to decode the message. Then it looks for listeners on this object and invokes your functions with the arguments to the message.

[19]https://en.wikipedia.org/wiki/Marshalling_(computer_science)

4. The Wayland display

Up to now we have discussed how wayland objects are jointly managed by the server and client. We have yet to describe how the object itself is created. The Wayland display, or `wl_display`, implicitly exists on every Wayland connection. It has the following interface:

```
<interface name="wl_display" version="1">
  <request name="sync">
    <arg name="callback" type="new_id" interface="wl_callback"
       summary="callback object for the sync request"/>
  </request>

  <request name="get_registry">
    <arg name="registry" type="new_id" interface="wl_registry"
      summary="global registry object"/>
  </request>

  <event name="error">
    <arg name="object_id" type="object" summary="object where the error occurred"/>
    <arg name="code" type="uint" summary="error code"/>
    <arg name="message" type="string" summary="error description"/>
  </event>

  <enum name="error">
    <entry name="invalid_object" value="0" />
    <entry name="invalid_method" value="1" />
    <entry name="no_memory" value="2" />
    <entry name="implementation" value="3" />
  </enum>

  <event name="delete_id">
    <arg name="id" type="uint" summary="deleted object ID"/>
  </event>
</interface>
```

The registry (`get_registry`) is used to allocate other objects. This chapter will cover how to use functions related to `wl_display`, which is less about the interface and protocol itself and more about the internals of `libwayland`.

4.1. Creating the display

This section explains how to use the `libwayland` implementation to create a display.

4.1.1. Wayland clients

Example code:

```c
#include <stdio.h>
#include <wayland-client.h>

int
main(int argc, char *argv[])
{
    struct wl_display *display = wl_display_connect(NULL);
    if (!display) {
        fprintf(stderr, "Failed to connect to Wayland display.\n");
        return 1;
    }
    fprintf(stderr, "Connection established!\n");

    wl_display_disconnect(display);
    return 0;
}
```

Compile with `-lwayland-client` flag and make sure to include the `wayland-client.h` file.

Establishing the connection and creating the display is done with `wl_display_connect(const char *name)`:

- `name` argument is the Wayland display (typically `wayland-0`), this corresponds to the name of a Unix socket in `$XDG_RUNTIME_DIR`[20].
  - `NULL` is preferred
- The following procedure describes how `libwayland` searches for the display socket:
  1. If `$WAYLAND_DISPLAY` is set, attempt to connect to `$XDG_RUNTIME_DIR/$WAYLAND_DISPLAY`
  2. Otherwise, attempt to connect to `$XDG_RUNTIME_DIR/wayland-0`

[20]This `env` variable is set at login, more information: https://askubuntu.com/questions/872792/what-is-xdg-runtime-dir

    3. Otherwise, fail.
- The user controls which display is used by changing the env variable `$WAYLAND_DISPLAY`.
- E.g.: `$ WAYLAND_DISPLAY=wayland-1 ./client`

You can also manually connect and create a Wayland display from a file descriptor:

```
struct wl_display *wl_display_connect_to_fd(int fd);
```

Obtaining the file descriptor of a `wl_display` is done with:

```
int wl_display_get_fd(struct wl_display *display);
```

The connection is closed with:

```
wl_display_disconnect(display);
```

4.1.2. Wayland servers

Example code:

```
#include <stdio.h>
#include <wayland-server.h>

int
main(int argc, char *argv[])
{
    struct wl_display *display = wl_display_create();
    if (!display) {
        fprintf(stderr, "Unable to create Wayland display.\n");
        return 1;
    }

    const char *socket = wl_display_add_socket_auto(display);
    if (!socket) {
        fprintf(stderr, "Unable to add socket to Wayland display.\n");
        return 1;
    }

    fprintf(stderr, "Running Wayland display on %s\n", socket);
    wl_display_run(display);

    wl_display_destroy(display);
    return 0;
}
```

The code is compiled with `-lwayland-server` and `wayland-server.h` header file.

The creation of the display and binding to a socket are separate, to give the server time to configure the display, before clients are able to connect to it.

This is done with the following two functions:

```
struct wl_display *display = wl_display_create();
const char *socket = wl_display_add_socket_auto(display);
```

`wl_display_add_socket_auto` will allow `libwayland` to decide the name for the display automatically. Which defaults to `wayland-0`, or `wayland-$n`, depending on whether any other Wayland compositors have sockets in `$XDG_RUNTIME_DIR`.

There are also more manual options:

```
int wl_display_add_socket(struct wl_display *display, const char *name);
int wl_display_add_socket_fd(struct wl_display *display, int sock_fd);
```

After adding the socket, calling `wl_display_run` will run `libwayland's` internal event loop and block until `wl_displat_terminate` is called.

The display is destroyed with:

```
wl_display_destroy(display);
```

4.2. Incorporating an event loop

`libwayland` provides its own event loop implementation for Wayland servers, and its considered out-of-scope. You do not need to use this event loop implementation, you can use a custom one.

4.2.1. Wayland server event loop

Each `wl_display` created by `libwayland` has its own corresponding `wl_event_loop`. A reference to this `wl_event_loop` can be obtained with `wl_display_get_event_loop`. A Wayland compositor will likely want to use this as its only event loop. You can add file descriptors, timers, signals to the event loop with:

```
wl_event_loop_add_fd()
wl_event_loop_add_timer()
wl_event_loop_add_signal()
```

Once the event loop is configured, you can process events and dispatch clients by calling `wl_display_run`, which will process the event loop and block until the display terminates. Most Wayland compositors use this approach.

However you can also monitor the event loop on your own, dispatching it as necessary, or you can disregard it entirely and manually process client updates. You can also treat the Wayland event loop as subservient (self managed) to your own event loop. This is done by using `wl_event_loop_get_fd` to obtain a *poll-able*[21] file descriptor, then call `wl_event_loop_dispatch` to process events when activity occurs on that file descriptor. You also need to call `wl_display_flush_clients` when you have data which needs writing to clients.

4.2.2. Wayland client event loop

`libwayland-client` does not have its own event loop. A simple loop suffices for one file descriptor instances:

```
while (wl_display_dispatch(display) != -1) {
    /* This space deliberately left blank */
}
```

However you can build your own event loop, by obtaining the Wayland display's file descriptor with `wl_display_get_fd`. Upon `POLLIN` events, call `wl_display_dispatch` to process incoming events. To flush outgoing requests, call `wl_display_flush`.

5. Globals and the registry

Each request and event is associated with an object ID. When we receive a Wayland message, we must know what interface the object ID represents to decode it. We must also negotiate available objects, the creation of new ones, and assign an ID to them, in some manner. In Wayland when we *bind* an object ID, we agree on the interface used for it in all future messages, and stash a mapping of objects IDs to interfaces in our local state.

The server offers a list of

1. global

objects, used for bootstrapping. Globals are most often used to broker additional objects to fulfill various purposes, such as the creation of windows. The globals themselves also have interfaces and object IDs, which also need to be assigned and agreed on.

This is solved by implicitly assigning object ID 1 to `wl_display` interface when you make the connection. The following is the interface for `wl_display`:

```
<interface name="wl_display" version="1">
  <request name="sync">
    <arg name="callback" type="new_id" interface="wl_callback" />
  </request>

  <request name="get_registry">
    <arg name="registry" type="new_id" interface="wl_registry" />
  </request>

  <!-- ... -->
</interface>
```

The `wl_display:get_registry` request can be used to bind an object ID to the `wl_registry` interface, which is the next one found in `wayland.xml`. It accepts one argument: **a generated ID for a new object**.

5.0.1. Wire message example

```
BIN: 00000001 000C0001 00000002
HEX: 0x01 0xC1 0x02
```

The first byte is the object ID, in this case ID $= 1$, which is the `wl_display` object. The most significant nibble in the second byte is the size of the message in bytes. The least significant nibble is the `opcode` in this case opcode $= 1$, the index of the request is tied to the interface, and also it's 0 indexed. In this case opcode `1` means the `get_registry` request defined in `wl_display` interface. The remaining byte is the argument for the request.

[21] https://pubs.opengroup.org/onlinepubs/009695399/functions/poll.html

Thus this wire message request operation with opcode `1` on the object with id `1` and passes in the argument `2`. The request could be written in human readable form as:

```
wl_display::get_registry(2)
```

Note that in the XML documentation this new ID is defined ahead of time to be governed by the `wl_registry` interface:

```
<interface name="wl_registry" version="1">
  <request name="bind">
    <arg name="name" type="uint" />
    <arg name="id" type="new_id" />
  </request>

  <event name="global">
    <arg name="name" type="uint" />
    <arg name="interface" type="string" />
    <arg name="version" type="uint" />
  </event>

  <event name="global_remove">
    <arg name="name" type="uint" />
  </event>
</interface>
```

5.1. Binding to globals

Upon creating a registry object, the server will emit the `global` event for each global available on the server. You can then bind the globals you require.

1. Binding

is the process of taking a known object and assigning it an ID.

Once the client binds to the registry like this, the server emits the `global` event several times to advertise which interfaces it supports.

Each global is assigned a unique `name` as a `uint`. The `interface` string maps to the name of the interface found in the protocol. The `version` number is also defined here.

Example: `<interface name="wl_registry" version="1">`.

To bind to any of these interfaces, we use the bind request. Consider the following wire protocol exchange:

```
C->S    00000001 000C0001 00000002            .... .... ....

S->C    00000002 001C0000 00000001 00000007   .... .... .... ....
        776C5f73 686d0000 00000001            wl_s hm.. ....
        [...]

C->S    00000002 00100000 00000001 00000003   .... .... .... ....
```

The first message is the same as the previous example. The second one is an event, object 2 (which the client assigned the `wl_registry` to), opcode 0 (`wl_registry::global`) with arguments `1`, `"wl_shm"` and `1`, respectively, the name, interface and version of this global. The client responds by calling opcode 0 on object ID 2 (`wl_registry::bind`) and assigns object ID 3 to global name 1, *binding* to the `wl_shm` global. Future events and requests for this object are defined by the `wl_shm` protocol (can be found in `wayland.xml`).

Once a object is created, it can be used for various tasks. The `wl_shm` object manages shared memory between the client and server.

5.2. Registering globals

Registering globals on the server side is done differently. `wayland-scanner` creates interfaces (analogous to listeners) and glue code for sending events. The first task is to register the global, with a function to rig up a *resource*[22] when the global is bound.

```
static void
wl_output_handle_bind(struct wl_client *client, void *data,
    uint32_t version, uint32_t id)
{
    struct my_state *state = data;
    // TODO
```

[22]The server-side state of each client's instance of an object

```c
}

int
main(int argc, char *argv[])
{
    struct wl_display *display = wl_display_create();
    struct my_state state = { ... };
    // ...
    wl_global_create(wl_display, &wl_output_interface,
        1, &state, wl_output_handle_bind);
    // ...
}
```

Add this code the example code in Section 4.1.2 and it makes `wl_output` global visible. However any attempts to bind to this global runs into our `TODO`. We need to provide an *implementation* of the `wl_output` interface as well.

```c
static void
wl_output_handle_resource_destroy(struct wl_resource *resource)
{
    struct my_output *client_output = wl_resource_get_user_data(resource);

    // TODO: Clean up resource

    remove_to_list(client_output->state->client_outputs, client_output);
}

static void
wl_output_handle_release(struct wl_client *client, struct wl_resource *resource)
{
    wl_resource_destroy(resource);
}

static const struct wl_output_interface
wl_output_implementation = {
    .release = wl_output_handle_release,
};

static void
wl_output_handle_bind(struct wl_client *client, void *data,
    uint32_t version, uint32_t id)
{
    struct my_state *state = data;

    struct my_output *client_output = calloc(1, sizeof(struct client_output));

    struct wl_resource *resource = wl_resource_create(
        client, &wl_output_interface, wl_output_interface.version, id);

    wl_resource_set_implementation(resource, &wl_output_implementation,
        client_output, wl_output_handle_resource_destroy);

    client_output->resource = resource;
    client_output->state = state;

    // TODO: Send geometry event, et al

    add_to_list(state->client_outputs, client_output);
}
```

We first extend our `bind` handler to create a `wl_resource` to track the server-side state for this object. We then provide `wl_resource_create` with a pointer to our implementation of the interface `wl_output_implementation`. This `struct` type is generated by `wayland-scanner` and contains one function pointer for each request supported by the interface. We also allocate a small container for storing any additional state we need that `libwayland` doesn't handle for us.

Note that `wl_output_interface` name is shared between the instance of an interface and a global constant variable generated by `wayland-scanner` which contains metadata related to the implementation (such as version above).

The `wl_output_handle_release` function is called when the client sends the `release` request, indicating they no longer need the resource and it can be destroyed. This triggers the `wl_output_handle_resource_destroy` function, which will free any of the state we allocated for it earlier. This function is passed into `wl_resource_create` as the destructor, and will be called if the client terminates without explicitly sending the `release` request.

The other remaining TODO is t send the name event, as well as a few others as per the wayland.xml file:

```
<event name="geometry">
  <description summary="properties of the output">
The geometry event describes geometric properties of the output.
The event is sent when binding to the output object and whenever
any of the properties change.

The physical size can be set to zero if it doesn't make sense for this
output (e.g. for projectors or virtual outputs).
  </description>
  <arg name="x" type="int" />
  <arg name="y" type="int" />
  <arg name="physical_width" type="int" />
  <arg name="physical_height" type="int" />
  <arg name="subpixel" type="int" enum="subpixel" />
  <arg name="make" type="string" />
  <arg name="model" type="string" />
  <arg name="transform" type="int" enum="transform" />
</event>
```

It is our responsibility to send this event when the output is bound:

```
static void
wl_output_handle_bind(struct wl_client *client, void *data,
    uint32_t version, uint32_t id)
{
    struct my_state *state = data;

    struct my_output *client_output = calloc(1, sizeof(struct client_output));

    struct wl_resource *resource = wl_resource_create(
        client, &wl_output_implementation, wl_output_interface.version, id);

    wl_resource_set_implementation(resource, wl_output_implementation,
        client_output, wl_output_handle_resource_destroy);

    client_output->resource = resource;
    client_output->state = state;

    wl_output_send_geometry(resource, 0, 0, 1920, 1080,
        WL_OUTPUT_SUBPIXEL_UNKNOWN, "Foobar, Inc",
        "Fancy Monitor 9001 4K HD 120 FPS Noscope",
        WL_OUTPUT_TRANSFORM_NORMAL);

    add_to_list(state->client_outputs, client_output);
}
```

Note that wl_output::geometry is shown here for illustrative purposes and you should review the XML before implementing this event in your client server.

6. Buffers and surfaces