RUSTY PIPES AND OXIDIZED WIRES

Arun Raghavan

ME

GStreamer, PipeWire, PulseAudio person

Previously: *Founder, Asymptotic*

Currently: *Engineer, Valve*

MOTIVATION

- PipeWire API is reasonably mature
- Rust bindings via FFI exist
- More safety would be nice
- Better ergonomics

FUNDING

- Massive shoutout to the *Sovereign Tech Agency*
- Would not have been possible without them

APPROACH

- Don’t try to rewrite everything
- Client protocol is a good starting point

REALITY

- Servers and clients aren’t *that* different
- But we can still be strategic

STRUCTURE: SPA

- Simple Plugin API
- A set of interfaces
- and their implementations

STRUCTURE: SPA

- Support interfaces
- Logging
- System
- Loops
- CPU features

STRUCTURE: SPA

- Other interfaces
- Device
- Node
- Codecs
- AEC

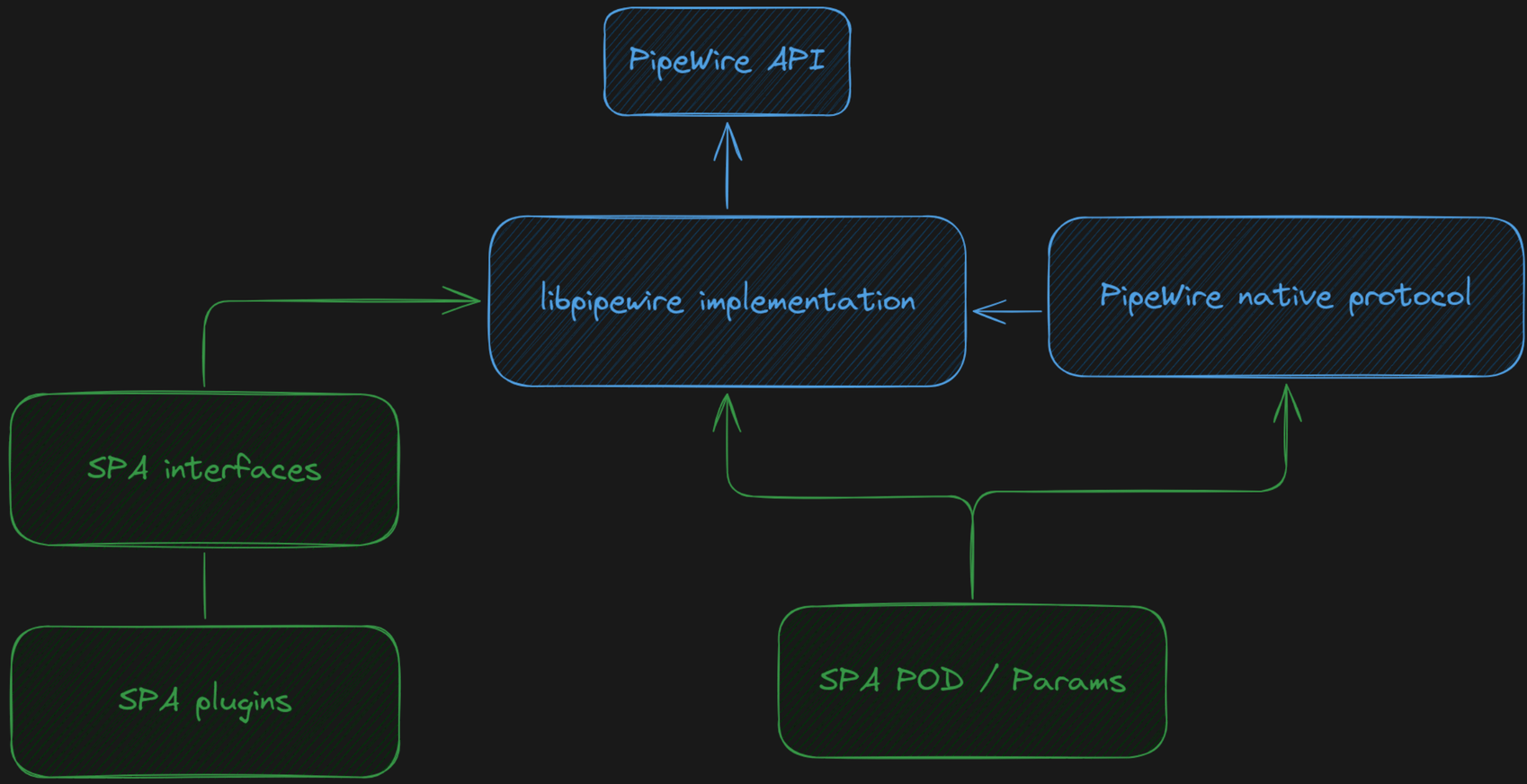
STRUCTURE: SPA

- Serialisation
- Objects
- Formats, params, …
- POD

STRUCTURE: PIPEWIRE

- Server
- Modules
- Native protocol
- Client library

STRUCTURE: PIPEWIRE + SPA



JUDICIOUS 

- SPA
  - Pod
  - Support *interfaces*
  - Reuse C plugins

JUDICIOUS 

- PipeWire
  - Client library
  - Native protocol
  - .so interop is tricky

EXAMPLE CODE

```rust
use pipewire_native as pw;

fn connect_to_pipewire() -> std::io::Result<()> {
    // Basic support initialisation
    pw::init();

    // A main loop for PipeWire communication
    let main_loop = pw::MainLoop::new(pw::Properties::new());

    // The context loads client configuration
    let context = pw::Context::new(&main_loop, pw::Properties::new())?;

    // Connecting provides a "core"
    let core = context.connect(None)?;
}
```

EXAMPLE CODE

```
fn listen_for_objects() -> std::io::Result<()> {
    // The registry notifies us about global objects
    let registry = core.registry()?;

    // To do that we connect a listener
    registry.add_listener(RegistryEvents {
        // registry is captured by cloning, the rest are arguments
        global: some_closure!([registry] id, perms, type_, ... {
            // Do something now that we know an object
            // of type_ was added
        },
        global_remove: some_closure!([] id, {
            // An object went away
        }
    });
}
```

EXAMPLE CODE

```
fn bind_device(registry: pw::Registry, id: u32, type_: &str,
    version: u32) -> std::io::Result<()> {

    // Tell the server we want to track this device
    let proxy = registry.bind(id, type_, version)?;

    // Downcast the generic proxy into the type we know it is
    let device =
        proxy.downcast::<pw::proxy::device::Device>().unwrap();

    // Call device-specific methods, and set up device-specific
    // event listeners
    ...
}
```

EXAMPLE CODE

```
pub fn link_nodes(&self, input: &pw::...::Node, output: &pw::...::Node) {
    // You can even create objects on the server
    let mut props = pw::Properties::new();
    props.set(
        pw::keys::LINK_INPUT_NODE,
        format!("{}", input.node.proxy().bound_id().unwrap()),
    );
    props.set(
        pw::keys::LINK_OUTPUT_NODE,
        format!("{}", output.node.proxy().bound_id().unwrap()),
    );
    self.core
        .create_object("link-factory",
            pw::types::interface::LINK, 3, &props)
        .unwrap();
}
```

LOTS TO DO

- Proxy and closure ergonomics
- Streaming
- Profiling

RUST

- Not all 🦄 and 🌈

RUST

- Lots of (circular) references
- Need strong and weak references
- Repeated “inner” pattern
- Also affects closure capture
- Maybe Ergonomic Rc will help

RUST

- Single- vs. multi-threaded
- Mutable borrows would be different
- Not an easy problem (duh)

RUST

- Lots to like
- Type system allows good modeling
- Safety by default
- Traits and macros reduce boilerplate
- New contributors

LINKS

- Crate: https://crates.io/crates/pipewire-native
- Tools: https://crates.io/crates/pipewire-native-tools
- Code: https://gitlab.freedesktop.org/pipewire/pipewire-native-rs
- Thanks to Sebastian Dröge for early reviews
- Feedback is welcome!

QUESTIONS?