Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Azalea

shell gráfico multicompositor para Wayland usando o protocolo layer shell

Enzo Hideo Kobayashi

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Alfredo Goldman
Cossupervisor: Prof. Nelson Posse Lago

São Paulo
2025

Resumo

Enzo Hideo Kobayashi. **Azalea: *shell gráfico multicompositor para Wayland usando o protocolo layer shell***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2025.

Shell gráfico é um programa responsável pelas interfaces gráficas que permitem um usuário interagir com o sistema operacional. Isso inclui as barras de tarefas, menu de início, papel de parede, barra de busca, menus de configuração, etc. Normalmente, os *shells* gráficos estão fortemente acoplados a um ambiente de área de trabalho, como o plasmashell do KDE e o GNOME Shell, tornando quase inviável trocar o compositor ou reusar o *shell* gráfico em outro ambiente. Nesse sentido, o objetivo deste projeto foi desenvolver um *shell* gráfico que funcione em múltiplos compositores Wayland diferentes por meio do protocolo *layer shell*.

O software foi desenvolvido em Rust usando Relm4 (GTK4) como biblioteca gráfica e Nix para criar ambientes de desenvolvimento reprodutíveis, empacotar a aplicação e testar a sua execução em diferentes compositores Wayland por meio de máquinas virtuais NixOS. O projeto também pode ser utilizado como bilioteca, permitindo que o usuário crie seus próprios componentes gráficos.

Palavras-chave: Linux. Wayland. protocolo *layer shell*. *shell* gráfico. Rust. GTK4. Relm4. Nix.

Abstract

Enzo Hideo Kobayashi. **Azalea: *compositor-agnostic graphical shell for Wayland using the layer shell protocol***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

Graphical shell is a program responsible for the graphical interfaces that allow a user to interact with the operating system. This includes taskbars, start menu, wallpaper, search bar, settings menus, etc. Usually, graphical shells are strongly coupled with a desktop environment, such as plasmashell from KDE and GNOME Shell, rendering it almost infeasible to replace the compositor or reuse the graphical shell in another environment. In view of that, the objective of this project is to develop a graphical shell that works in multiple different Wayland compositors through the layer shell protocol.

The software was developed in Rust using Relm4 (GTK4) as a graphical library and Nix to create reproducible development environments, package the application, and test its execution in different Wayland compositors using NixOS virtual machines. The project can be used as a library as well, allowing the user to create their own graphical components.

Keywords: Linux. Wayland. layer shell protocol. graphical shell. Rust. GTK4. Relm4. Nix.

Lista de abreviaturas

D-Bus	Desktop Bus
MPRIS	Media Player Remote Interfacing Specification
MVU	Model-View-Update
LSP	Language Server Protocol
RAII	Resource acquisition is initialization
UUID	Universally Unique Identifier
SASS	Syntactically Awesome Style Sheets
CSS	Cascading Style Sheets
JSON	JavaScript Object Notation
RON	Rusty Object Notation
ALSA	Advanced Linux Sound Architecture
ACPI	Advanced Configuration and Power Interface
DDC/CI	Display Data Channel/Command Interface

Lista de figuras

2.1 Lista de compositores com suporte ao protocolo *layer shell* (Wayland Explorer, 2025) . . . 4
2.2 Representação visual das camadas e âncoras do protocolo layer shell . . . 4

4.1 Papel de parede padrão . . . 18
4.2 *Slider* de volume . . . 20
4.3 *Slider* de brilho . . . 20
4.4 *Widget* de data e hora . . . 21
4.5 *Widget* do menu de início. . . . 22
4.6 *Widget* de controle de dispositivos *bluetooth* . . . 22
4.7 *Widget* de controle de conexões rede . . . 23
4.8 *Widget* de notificações . . . 24
4.9 *Widget* MPRIS . . . 25
4.10 *Widgets* de *shortcut* para Steam, Chromium e Firefox . . . 25
4.11 *Widget* de busca de aplicações . . . 26
4.12 Colagem de fotos com todos os *widgets* . . . 27

5.1 Azalea rodando no Sway em máquina virtual NixOS. . . . 31
5.2 Azalea rodando no Hyprland em máquina virtual NixOS. . . . 31
5.3 Azalea rodando no Wayfire em máquina virtual NixOS. . . . 32
5.4 Azalea rodando no Miracle em máquina virtual NixOS. . . . 32
5.5 Azalea rodando no Niri em máquina virtual NixOS. . . . 35

Lista de programas

3.1 Exemplo de macros procedurais em Rust. 8
3.2 Exemplo de macro simples em Rust. 9
3.3 Exemplo básico de uso da biblioteca gtk4-layer-shell 10
3.4 Exemplo de uso da biblioteca Relm4 para fazer um *widget* "contador" de apenas um botão, adaptado de *GUI development with Relm4*, Cap. 2 (Relm4 contributors, 2025). 11
3.5 Exemplo de *proxy* usando zbus. 12
3.6 Exemplo de daemon de notificação implementado com zbus. 12
3.7 Exemplo de estilo Sass usado no Azalea. 13
3.8 Exemplo de uso da macro *derive StaticServiceManager* para implementar um gerenciador de serviço *singleton*. 15
4.1 Exemplo de configuração do papel de parede em formato RON. 18
4.2 Exemplo de configuração da barra de tarefas do canto inferior da tela em formato RON. 19
4.3 Exemplo de configuração do *widget* de data e hora RON. 21
4.4 Exemplo de configuração de *shortcut* em formato RON. 25
5.1 Exemplo de como usar um *dev shell* nix. 29
5.2 Exemplo de *dev shell* nix. 30
5.3 Exemplo simples de como empacotar uma aplicação fantasia em nix. . . 33
5.4 Exemplo de teste em máquina virtual NixOS 34

Sumário

1 Introdução **1**

2 Conceitos fundamentais **3**

2.1 Shell . . . 3

2.2 Wayland . . . 3

2.2.1 Protocolo *layer shell* . . . 4

2.3 D-Bus . . . 5

3 Detalhes de implementação **7**

3.1 Ferramentas . . . 7

3.1.1 Rust . . . 7

3.1.2 Relm4, GTK4 e gtk4-layer-shell . . . 8

3.1.3 Zbus . . . 10

3.1.4 Sass . . . 10

3.2 Arquitetura . . . 10

3.2.1 Cliente-Servidor . . . 10

3.2.2 Serviços . . . 13

4 Funcionalidades de *desktop* **17**

4.1 Papel de parede . . . 18

4.2 Barra de tarefas . . . 19

4.2.1 Controle de volume . . . 20

4.2.2 Controle de brilho . . . 20

4.2.3 Data e hora . . . 20

4.2.4 Menu de início . . . 21

4.2.5 Controle de dispositivos *bluetooth* . . . 21

4.2.6 Controle de conexões de rede . . . 22

4.2.7 Notificações . . . 24

4.2.8 Controle de mídia . . . 24

4.2.9 *Shortcuts* . . . 25
4.2.10 Barra de busca de aplicações . . . 26
4.2.11 *System tray* . . . 26
4.3 *Overview* . . . 27

5 Nix **29**
5.0.1 Ambiente de desenvolvimento . . . 29
5.0.2 Empacotamento . . . 29
5.0.3 Testes em máquinas virtuais NixOS . . . 30

6 Conclusão **37**

Referências **39**

Capítulo 1

Introdução

Um dos pontos fortes do sistema operacional Linux é a sua modularidade, no sentido de permitir ao usuário escolher e trocar vários componentes do sistema. Desde gerenciadores de login, ambientes de área de trabalho, gerenciadores de janela, shells de linha de comando, até o sistema de *init*.

No entanto, o mesmo não pode ser necessariamente dito dos shells gráficos. Há um forte acoplamento entre o shell gráfico e o gerenciador de janela, ou compositor. Por exemplo, não é possível utilizar o GNOME Shell em outro ambiente que não seja o do Mutter, o compositor Wayland do GNOME.

Nesse sentido, o objetivo deste trabalho é desenvolver um shell gráfico que funcione em vários compositores Wayland diferentes. E o programa deve implementar funcionalidades essenciais de um *desktop*, como:

- Papel de parede
- Barra de tarefas customizável
- Controle de volume
- Controle de brilho
- Controle de dispositivos bluetooth
- Controle de conexões de rede
- Data e hora
- Notificações
- Controle de mídia
- Menu de início
- *Shortcuts* para abrir aplicações
- Barra de busca de aplicações
- *System tray*

O próximo capítulo explica conceitos fundamentais para compreender o projeto. O terceiro capítulo trata das ferramentas utilizadas e detalhes gerais de implementação. O quarto capítulo fala sobre as funcionalidades implementadas. O quinto capítulo trata do uso do Nix no projeto. Por fim, o último capítulo conclui com os resultados alcançados e outros protocolos Wayland de interesse que não foram tratados neste trabalho.

Capítulo 2

Conceitos fundamentais

O programa se trata de um *shell* gráfico para Wayland que utiliza o protocolo *layer shell* para funcionar em múltiplos compositores Wayland e usa D-Bus para a implementação de diversas funcionalidades de um *desktop*.

Este capítulo é dedicado a explicar cada um desses termos e discutir as ferramentas utilizadas.

2.1 Shell

O *shell* é um programa que permite ao usuário interagir com o sistema, atuando como uma "camada" entre o usuáro e o sistema operacional (Wikipedia contributors, 2025).

Existem *shells* de linha de comando como bash e zsh, mas também existem os *shells* gráficos como o plasmashell, GNOME Shell e este projeto, o Azalea. O *shell* gráfico é responsável, basicamente, por todos as interfaces gráficos usadas para interagir com o sistema.

2.2 Wayland

Wayland é um protocolo de display criado para substituir o X11 (Freedesktop, 2025b). No X11, o compositor e gerenciador de janelas são programas distintos, porém no caso do Wayland o compositor é responsável tanto pelo gerenciamento de janelas quanto da composição. Isto é, o compositor é responsável por decidir o posicionamento das janelas e pode aplicar efeitos, como shaders e animações.

O compositor Wayland atua como servidor de display e cada aplicação que deseja criar uma janela atua como cliente. Os protocolos Wayland especificam como essa comunicação entre cliente e servidor deve acontecer. Existem os protocolos *core*, que todo compositor Wayland deve implementar, e existem os protocolos de extensão, que são opcionais e extendem a funcionalidade do compositor. Um desses protocolos de extensão é o protocolo *layer shell* ou, mais precisamente, wlr-layer-shell-unstable-v1.

2.2.1 Protocolo *layer shell*

Protocolos Wayland passam por três fases principais: experimental ou *unstable*, *staging* e *stable*. A fase experimental é a fase em que o protocolo ainda está em desenvolvimento. Quando o protocolo alcança certa maturidade e reconhecimento, ele passa para a fase de teste ou *staging* em que ele é incluído no repositório principal (https://gitlab.freedesktop.org/wayland/wayland-protocols) e os compositores e clientes Wayland são incentivados a implementar o protocolo para fins de teste. Por fim, após ser suficientemente testado pela comunidade, ele pode ser considerado estável (FREEDESKTOP, 2025c).

O protocolo *layer shell* é experimental e surgiu inicialmente no wlroots, uma biblioteca de criação de compositores Wayland. Já que essa biblioteca é utilizada por vários compositores, o protocolo *layer shell* é amplamente suportado apesar de ser experimental (2.1).

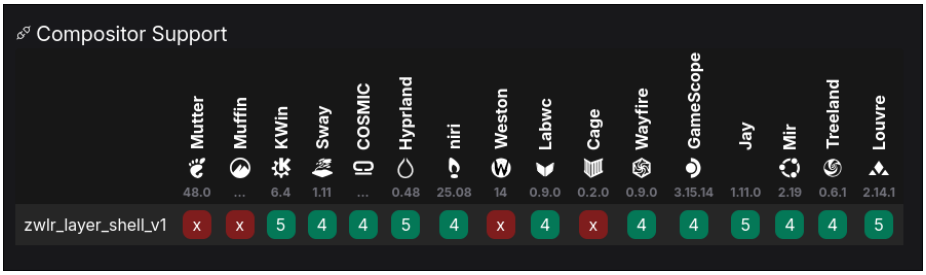


Figura 2.1: *Lista de compositores com suporte ao protocolo layer shell (WAYLAND EXPLORER, 2025)*

Esse protocolo permite que uma aplicação cliente peça ao compositor em qual camada a sua janela deve ser renderizada e em qual canto da tela deve ser ancorada (2.2).

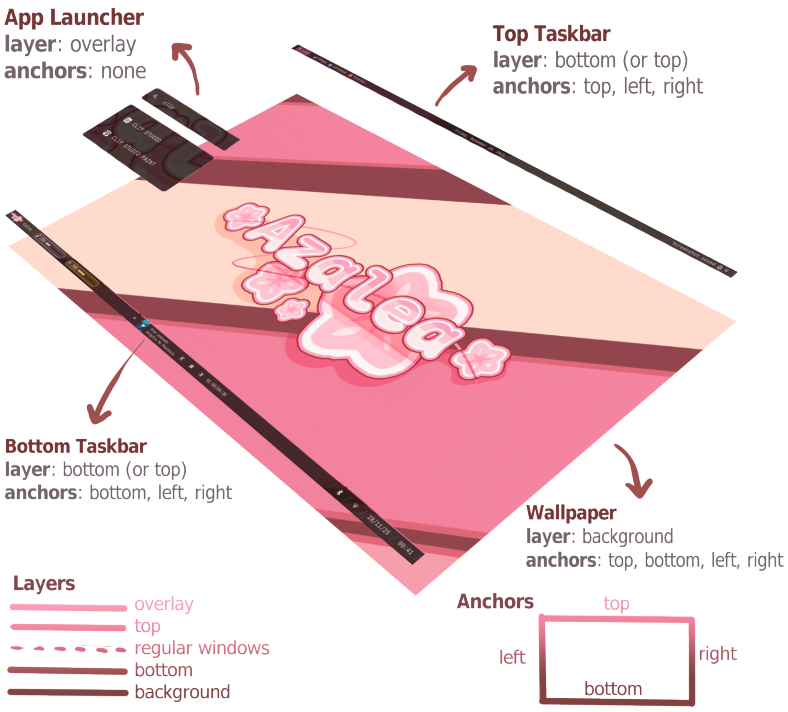


Figura 2.2: *Representação visual das camadas e âncoras do protocolo layer shell*

O cliente também pode pedir ao compositor uma "zona exclusiva" para evitar que sua janela intersecte outras janelas. Isso é particularmente útil em barras de tarefas para evitar que ela sejam sobrepostas por outras janelas. No entando, esse comportamento depende da implementação, então o resultado pode ser um pouco diferente dependendo do compositor.

2.3 D-Bus

D-Bus, ou *Desktop Bus*, é um sistema de barramento de mensagens que permite comunicação entre processos. Seu principal uso, como o nome sugere, é conectar os diversos components e aplicações utilizados em um *desktop*.

Normalmente, são utilizados dois daemons de barramento: sistema e sessão. O barramento de sistema é utilizado para comunicação que de processos que abrange o sistema como um todo, como o *Network Manager*. Enquanto que o barramento de sessão é utilizado apenas na sessão do usuário, como o MPRIS.

Aplicações clientes D-Bus se conectam a um desses barramentos criando uma conexão de nome único. Essa conexão pode expor vários "objetos" e cada objeto pode implementar várias interfaces, que definem os métodos, propriedades e sinais que o objeto expõe. Os métodos permitem receber e retornar valores numéricos, strings, arrays, dicionários e outras estruturas mais complexas. As propriedades são valores do objeto que podem permitir leitura e escrita, apenas leitura, ou apenas escrita. Os sinais de um objeto permitem uma comunicação de tipo *publish-subscribe*, em que os objetos podem fazer *broadcast* de um sinal e clientes interessados podem se inscrever para serem alertados quando esse sinal é emitido (Freedesktop, 2025a).

Em termos de implementação, há dois tipos principais de uso do D-Bus: os serviços e os clientes, ou *proxies*. Uma aplicação serviço é aquela que se conecta ao barramento com um nome específico e fornece objetos que implementam alguma interface. Aplicações clientes que desejam usar esse serviço precisam implementar um *proxy*, uma estrutura de dados local no programa que representa esse objeto e implementa suas interfaces, de modo que uma chamada de método nesse *proxy* corresponderia a chamar o método do objeto D-Bus.

Capítulo 3

Detalhes de implementação

3.1 Ferramentas

O programa foi escrito em Rust utilizando Relm4 (GTK4) como biblioteca gráfica, gtk4-layer-shell para usar o protocolo *layer shell* e zbus para implementar os serviços e *proxies* D-Bus. Outras bibliotecas de usos específicos são mencionadas nas seções seguintes quando tal uso for explicado.

3.1.1 Rust

Rust (https://rust-lang.org/) é uma linguagem de programação de tipagem forte e estática. Um de seus pontos principais é a segurança de memória, alcançada por meio de seu sistema de *ownership*: todo valor possui um único dono e quando esse dono sai do escopo, o valor é destruído.[1] Nesse sentido, Rust não possui um coletor de lixo, a memória é gerenciada em tempo de compilação de forma similar à técnica de RAII.

Uma funcionalidade Rust utilizada amplamente no projeto são as macros procedurais de tipo *derive* e *attribute*, que permitem a implementação automática de *traits*[2] e atributos customizados, respectivamente. O exemplo em 3.1 demonstra o uso da macro *derive* para implementar o *trait Default* e uma macro atributo para anotar qual membro da enumeração deve ser usado como valor padrão.

Também existem as macros não-procedurais, que atuam como uma função, mas permitem argumentos de tipos diferentes e quantidades variáveis de argumentos. O exemplo 3.2 demonstra um uso básico em que a macro `my_macro` aceita ser chamada de três formas diferentes:

- `my_macro!(arg: literal)`: Imprime o literal
- `my_macro!(arg1: literal, arg2: literal)`: Imprime os dois literais

[1] Uma variável pode ser destruída antes que ela saia de escopo por meio da função *drop*. *The Rust Programming Language, Third Edition*, Cap. 4 (Steve Klabnik, Carol Nichols, Chris Krycho, Rust community contributors, 2025)

[2] *Traits* em Rust são como interfaces, eles definem quais métodos devem ser implementados

- `my_macro!(arg: type)`: Implementa o *trait Display* para um tipo qualquer

Programa 3.1 Exemplo de macros procedurais em Rust.

```rust
// Macro derive que implementa os traits Default, Debug e PartialEq
#[derive(Default, Debug, PartialEq)]
enum Animal {
    // Macro atributo que anota este valor como padrão
    //
    // Isso é usado pela macro derive para implementar o trait Default
    #[default]
    Cat,

    Dog,
}

// A implementação manual do trait Default seria:
// impl Default for Animal {
//  fn default() -> Self {
//   Animal::Cat
//  }
// }

fn main() {
    // Já que o trait Default foi implementado, podemos usar a função
    // default() para criar um Animal padrão.
    let animal = Animal::default();

    // Verifica que o animal criado é de fato o valor padrão Animal::Cat
    assert_eq!(animal, Animal::Cat); // Necessita os traits Debug e PartialEq
}
```

3.1.2 Relm4, GTK4 e gtk4-layer-shell

GTK4 (https://www.gtk.org/) é um *toolkit* de *widgets* de código aberto do GNOME que permite a criação de aplicações gráficas. A biblioteca é *single-threaded*, no sentido que o laço principal que cuida dos eventos da aplicação e os *widgets* devem estar na mesma thread, não é possível enviar objetos GTK entre *threads*.

Relm4 (https://relm4.org/) é uma biblioteca que simplifica o uso do GTK4 em Rust com base na arquitetura Elm (*Model-View-Update*). Essa arquitetura consiste em três componentes principais:

- *Model*: conjunto de dados que define o estado do *widget*
- *View*: representação gráfica do modelo
- *Update*: define como o modelo deve ser atualizado

O exemplo 3.4 demonstra como essa arquitetura é utilizada para criar *widgets*.

A biblioteca gtk4-layer-shell adiciona supporte ao protocolo *layer shell* ao GTK4. O exemplo 3.3 mostra como configurar as camadas, âncoras e zona exclusiva nas janelas criadas pelo GTK.

Programa 3.2 Exemplo de macro simples em Rust.

```rust
// Declaração da macro
#[macro_export]
macro_rules! my_macro {
    ($arg:literal) => {
        println!("Literal: {}", $arg);
    };
    ($arg1:literal, $arg2:literal) => {
        println!("Two literals: {} {}", $arg1, $arg2);
    };
    ($arg:ty) => {
        impl std::fmt::Display for $arg {
            fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
                write!(f, "Generic display: {}", std::any::type_name::<Self>())
            }
        }
    };
}

// Enumeração com valores internos
enum Color {
    RGB(u8, u8, u8),
    BGR(u8, u8, u8),
    RGBA(u8, u8, u8, u8),
    ComplexRgb {
      r: u8,
      g: u8,
      b: u8,
    }
}

// Essa chamada implementa o trait de Display para a struct Color
my_macro!(Color);

fn main() {
    my_macro!("hello world");
    my_macro!(3, 3);

    // O trait de Display permite que instâncias da struct sejam transformadas
    // em string e imprimidas na tela pela macro println
    println!("{}", Color::RGB(0, 0, 255));

    // output:
    //
    // Literal: hello world
    // Two literals: 3 3
    // Generic display: main::Color
}
```

Programa 3.3 Exemplo básico de uso da biblioteca gtk4-layer-shell

```
1  let window = gtk::Window::new();
2
3  window.init_layer_shell();
4  window.set_namespace(Some("taskbar"));
5  window.set_keyboard_mode(gtk4_layer_shell::KeyboardMode::OnDemand);
6  window.set_exclusive_zone(-1);
7  window.set_layer(gtk4_layer_shell::Layer::Overlay);
8  window.set_anchor(gtk4_layer_shell::Edge::Top, true);
```

3.1.3 Zbus

Zbus (https://z-galaxy.github.io/zbus/) é uma biblioteca em Rust que permite a criação de serviços e proxies D-Bus por meio de macros procedurais. O programa 3.5 mostra um exemplo de *proxy* para a interface D-Bus MPRIS e 3.6 mostra a implementação de um daemon de notificações. As macros procedurais tornam a implementação consideravelmente mais simples.

3.1.4 Sass

Sass (https://sass-lang.com/) é uma linguagem que extende o CSS e foi utilizada no projeto para estilizar visualmente o programa. Já que a biblioteca GTK4 já possui suporte para CSS, bastou compilar o arquivo de `Sass` para `CSS`. Para isso, foi utilizada a biblioteca *grass* (https://crates.io/crates/grass).

O usuário pode carregar um estilo customizado por meio do arquivo de configuração em "*HOME/.config/azalea/style.scss*" (3.7).

3.2 Arquitetura

3.2.1 Cliente-Servidor

O programa possui dois modos de execução: servidor e cliente, dependendo do comando:

- `azalea daemon start`: Inicia o servidor
- `azalea daemon stop`: Para o servidor
- `azalea window uuid`: Lista os UUIDs de todos os *widgets*
- `azalea window toggle [UUID]`: Mostra ou esconde um *widget*
- `azalea window create [NOME]`: Cria um *widget* a partir de seu nome
- `azalea layer toggle [NAMESPACE]`: Mostra ou esconde todos os *widgets* em camadas pertencentes a um certo *namespace*
- `azalea style reload`: Recarregar estilo Sass

Programa 3.4 Exemplo de uso da biblioteca Relm4 para fazer um *widget* "contador" de apenas um botão, adaptado de *GUI development with Relm4*, Cap. 2 (RELM4 CONTRIBUTORS, 2025).

```
1   struct Model {
2       counter: u8,
3   }
4
5   enum Input {
6       Increment,
7   }
8
9   #[relm4::component]
10  impl relm4::SimpleComponent for Model {
11      type Init = u8; // dados de inicialização do modelo
12      type Input = Input;
13      type Output = ();
14
15      view! {
16          gtk::Window {
17              gtk::Button {
18                  // Quando o botão é clicado, a mensagem de Input::Increment é
                        gerada
19                  connect_clicked => Input::Increment
20
21                  // Quando o Model atualizar, este valor também será atualizado
22                  #[watch]
23                  set_label: &format!("Counter: {}", model.counter),
24              },
25          }
26      }
27
28      fn update(&mut self, message: Self::Input, _sender: ComponentSender<Self>)
            {
29          match message {
30              Input::Increment => {
31                  self.counter = self.counter.wrapping_add(1);
32              }
33          }
34      }
35
36      // Construtor
37      fn init(
38          init: Self::Init,
39          root: Self::Root,
40          sender: ComponentSender<Self>,
41      ) -> ComponentParts<Self> {
42          let model = Model { counter: init };
43          let widgets = view_output!();
44          ComponentParts { model, widgets }
45      }
46  }
```

Programa 3.5 Exemplo de *proxy* usando zbus.

```rust
#[derive(Clone, Debug, Default, serde::Serialize, serde::Deserialize,
    OwnedValue)]
#[zvariant(signature = "s")]
pub enum PlaybackStatus {
   Playing,
   Paused,
   #[default]
   Stopped,
}

#[proxy(
   interface = "org.mpris.MediaPlayer2.Player",
   default_path = "/org/mpris/MediaPlayer2"
)]
pub trait Player {
   // Método do objeto D-Bus
   fn play_pause(&self) -> zbus::Result<()>;

   // Propriedade do objeto D-Bus
   #[zbus(property)]
   fn playback_status(&self) -> zbus::fdo::Result<PlaybackStatus>;

   // Sinal emitido pelo objeto D-Bus
   #[zbus(signal)]
   fn seeked(&self, position: i64) -> zbus::Result<()>;
}
```

Programa 3.6 Exemplo de daemon de notificação implementado com zbus.

```rust
#[derive(Debug, zbus::zvariant::Type, serde::Serialize, serde::Deserialize)]
pub struct ServerInformation {
   pub name: String,
   pub vendor: String,
   pub version: String,
   pub spec_version: String,
}

pub struct Notifications { }

/// https://specifications.freedesktop.org/notification/1.3/protocol.html
#[zbus::interface(name = "org.freedesktop.Notifications")]
impl Notifications {
   fn get_server_information(&self) -> zbus::fdo::Result<ServerInformation> {
      Ok(ServerInformation {
         name: format!(env!("CARGO_PKG_NAME")),
         vendor: format!(env!("CARGO_PKG_AUTHORS")),
         version: format!(env!("CARGO_PKG_VERSION")),
         spec_version: format!("1.3"),
      })
   }
}
```

Programa 3.7 Exemplo de estilo Sass usado no Azalea.

```
:root {
  --background: #191112;
  --error: #ffb4ab;
  --error-container: #93000a;
  --on-background: #f0dee0;
  --on-error: #690005;
  --on-error-container: #ffdad6;
  --on-primary: #561d2a;
  --on-primary-container: #ffd9de;
  --on-secondary: #43292d;
  --on-secondary-container: #ffd9de;
  --on-surface: #f0dee0;
  --on-tertiary: #452b08;
  --on-tertiary-container: #ffddba;
  --primary: #ffb2bd;
  --primary-container: #72333f;
  --secondary: #e5bdc1;
  --secondary-container: #5c3f43;
  --surface: #191112;
  --surface-container: #261d1e;
  --tertiary: #eabf90;
  --tertiary-container: #5f411c;
}

* {
  color: var(--on-surface);
  border: none;
  outline: none;
}

popover {
  contents,
  arrow {
    background: unquote("mix(var(--surface), transparent, 0.2)");
  }
}
```

O servidor é responsável por iniciar a aplicação GTK e ouve por conexões em uma *socket unix*. A sua responsabilidade é executar o laço de eventos do GTK e gerenciar a criação dos *widgets*.

O cliente permite enviar mensagens ao servidor pela linha de comando. Isso é particularmente útil se o usuário quiser, por exemplo, recarregar um estilo customizado Sass sem ter que reiniciar o programa ou esconder temporariamente a barra de tarefas.

3.2.2 Serviços

Serviço é uma abstração formada por dois componentes: a implementação da funcionalidade central do serviço e o gerenciador.

A implementação do serviço é feita utilizando um trait chamado `Service` que possui

as seguintes responsabilidades:

- Tratar mensagens de *Input* recebida dos clientes
- Enviar mensagens de *Output* para os clientes
- Gerar eventos internos (opcional)
- Tratar eventos internos (opcional)

As mensagens de *Input* atuam como ações que os clientes mandam para o serviço e as mensagens de *Output* indicam mudanças de estado do serviço.

O gerenciador é uma estrutura de dados genérica que atua como *wrapper* em torno do serviço. O gerenciador é responsável por:

- *stop()*: Iniciar o serviço
- *start()*: Parar o serviço
- *message(Input)*: Enviar mensagem para o serviço
- *listen(...)*: Permitir que clientes se inscrevam para receber mensagens de *Output*

Para facilitar o uso dos serviços e gerenciadores, foi implementada uma macro de tipo *derive* que cuida da criação de um gerenciador *singleton* e permite que funções do gerenciador sejam chamadas diretamente na *struct* do serviço como métodos estáticos. Assim, o usuário só precisa se preocupar em implementar o *trait* de serviço e a macro cuida do gerenciador. O exemplo 3.8 demonstra o seu uso em comparação à criação manual do gerenciador.

Nesse sentido, se quisermos criar um serviço que controla os dispositos bluetooth, poderíamos ter algo como:

- *Input*: ligar/desligar bluetooth, conectar/desconectar dispositivo
- *Output*: estado de dispositivo bluetooth
- Tratar mensagens: recebe *Input* e altera o estado de algum dispositivo bluetooth
- Gerar eventos internos: ouve por mudanças no estado dos dispositivos bluetooth
- Tratar eventos internos: se estado do dispositivo mudou, envia mensagem de *Output*

Com isso, é possível usar essa abstração de serviço para descrever todas as funcionalidades de *desktop* do programa, desde controle de áudio e brilho até *daemons* de notificação D-Bus. O próximo capítulo aprofunda cada uma dessas funcionalidades.

Note também que o design dessa abstração de serviço é similar ao MVU usado pelo Relm4, porém sem o *View*. Isso permite que o *Model-Update* (o serviço) e o *View* (os *widgets* GTK) rodem em *threads* diferentes. Assim, o programa faz uso de múltiplas *threads* mesmo que o GTK seja *single-threaded*. Outra vantagem é que múltiplos *widgets* podem fazer uso de um mesmo serviço, isto é, uma comunicação de tipo `1:N` (*publish-subscribe*).

Programa 3.8 Exemplo de uso da macro *derive StaticServiceManager* para implementar um gerenciador de serviço *singleton*.

```
// Esta macro derive adiciona métodos estáticos ao ServiceA
// que são delegados ao gerenciador singleton
#[derive(azalea_derive::StaticServiceManager)]
pub struct ServiceA {
  pub data: String
}

impl azalea_service::Service for ServiceA {
    type Init = (); // dados de inicialização do servidor

    type Input = ();
    type Event = ();
    type Output = ();

    // ...
}

fn main() {
  // A macro derive permite chamar os métodos estaticamente
  // na struct do serviço
  ServiceA::start();
  ServiceA::stop();

  // Sem a macro, seria necessário criar uma instância do gerenciador
  // e emprestar essa referência a TODOS os widgets que fazem uso
  // desse serviço...
  //
  // Ou criar teria que criar o singleton manualmente...
  //
  let serviceA_manager = ServiceManager::<ServiceA>::new(...);
  serviceA_manager.start();
  serviceA_manager.stop();

  // OBS.: Quando um cliente deseja ouvir as mensagens de Output,
  // ele recebe um "ListenerHandle". O conceito de Ownership é
  // utilizado aqui para automaticamente fechar a conexão se
  // esse handle sair de escopo.
  let listen_handle = ServiceA::listen(...);
}
```

Capítulo 4

Funcionalidades de *desktop*

Este capítulo trata das funcionalidades de *desktop* implementadas, ou seja, os *widgets* e serviços. Cada *widget* possui uma estrutura de configuração que permite configurar o *widget* a partir de um arquivo em formato JSON ou RON.[1] Para fazer a serialização e desserialização desse arquivo de configuração foi utilizada a biblioteca serde (https://serde.rs/).

A configuração de uma janela, como o papel de parede (4.1) ou barra de tarefas (4.2), possui a seguinte estrutura:

- `config`: configuração específica do *widget*
- `layer_shell`: permite configurar a camada, âncora, *namespace* e zona exclusiva
- `lazy`: Indica que o programa não deve criar esta janela durante a inicialização (isso só é util se o usuário precisar de uma janela criada dinamicamente com o comando "`azalea window create [NOME]`")
- `monitor`: Indica em qual monitor a janela deve ser criada

[1] RON (https://docs.rs/ron/latest/ron/) é uma notação cuja sintaxe se parece com código Rust

4.1 Papel de parede

O papel de parede é simplesmente uma janela que mostra uma imagem na camada de *background* e é ancorada aos quatro cantos da tela. Sua configuração também é simples, pois precisa receber apenas o caminho da imagem. Se não for especificado o caminho de alguma imagem, o programa usa o papel de parede padrão (4.1).[2]

Figura 4.1: *Papel de parede padrão*

Programa 4.1 Exemplo de configuração do papel de parede em formato RON.

```
"wallpaper": (
   config: Wallpaper(
      image: None,
   ),
   layer_shell: (
      namespace: "wallpaper",
      layer: Background,
      anchors: [
         Left,
         Right,
         Bottom,
         Top,
      ],
      exclusive_zone: Ignore,
   ),
   lazy: false,
   monitor: All,
)
```

[2] ilustração feita pelo autor

4.2 Barra de tarefas

A barra de tarefas implementada permite customizar a ordem dos widgets na barra. A barra é dividida em 3 áreas: início, centro e fim; e cada área pode possuir múltiplos *widgets*. O exemplo 4.2 mostra a configuração para uma barra de tarefas do canto inferior da tela.

Note que a partir do arquivo de configuração é extremamente fácil mover essa barra de tarefas para o topo, basta mudar a âncora da propriedade *layer_shell* e recarregar o programa.

Programa 4.2 Exemplo de configuração da barra de tarefas do canto inferior da tela em formato RON.

```
"bottom-taskbar": (
   config: Taskbar(
      spacing: 8,
      start: [
         StartMenu(),
         Audio(),
         Brightness(),
      ],
      center: [
         Media(),
      ],
      end: [
         Bluetooth(),
         Network(),
         Separator(separator: None),
         Time(
            format: "%d/%m/%y",
         ),
         Separator(separator: None),
         Time(
            format: "%H:%M",
         ),
      ],
   ),
   layer_shell: (
      namespace: "taskbar",
      layer: Top,
      anchors: [
         Left,
         Right,
         Bottom, # canto inferior da tela
      ],
      exclusive_zone: Auto,
   ),
   lazy: false,
   monitor: All,
),
```

Várias das implementações nas subseções seguintes utilizam a abstração de serviço explicada na seção 3.2.2. Em suma, os serviços podem ser descritos pelos tipos de mensagem

que ele recebe dos clientes (*Input*) e as mensagens que envia por *broadcast* a todos os clientes (*Output*). Em casos especiais há também eventos internos, mas os clientes (*widgets*) não precisam saber disso para usar o serviço.

4.2.1 Controle de volume

O controle de volume foi implementado utilizando ALSA e possui mensagens bem simples:

- `Input::SystemVolume(f64)`: Requisição para mudar o volume
- `Output::SystemVolume(f64)`: Novo valor do volume *broadcasted* para os clientes

O *widget* que usa esse serviço consiste em um *slider* que controla o volume (4.2).

Figura 4.2: *Slider de volume*

4.2.2 Controle de brilho

O controle de brilho é semelhante ao de áudio, porém utilizando a biblioteca *brightness* (https://crates.io/crates/brightness), que permite controlar o brilho por ACPI dos dispositivos em "`/sys/class/backlight`". Isso funciona bem em telas de notebook. Para monitores externos que utilizam DDC/CI seria necessário ter o driver *ddcci-backlight* (https://gitlab.com/ddcci-driver-linux/ddcci-driver-linux).

- `Input::SystemBrightness(f64)`: Requisição para mudar o brilho
- `Output::SystemBrightness(f64)`: Novo valor do volume *broadcasted* para os clientes

Analogamente ao *widget* anterior, o controle de brilho também foi implementado como um *slider* (4.3).

Figura 4.3: *Slider de brilho*

4.2.3 Data e hora

Este *widget* recebe o formato da data por meio da configuração (4.3) e mostra na tela a data e/ou hora. Ao clicar nessa data, ele cria um *popup* com o calendário padrão da biblioteca GTK (4.4).

Programa 4.3 Exemplo de configuração do *widget* de data e hora RON.

```
Time(
  format: "%A, %B %d, %Y",
)
```

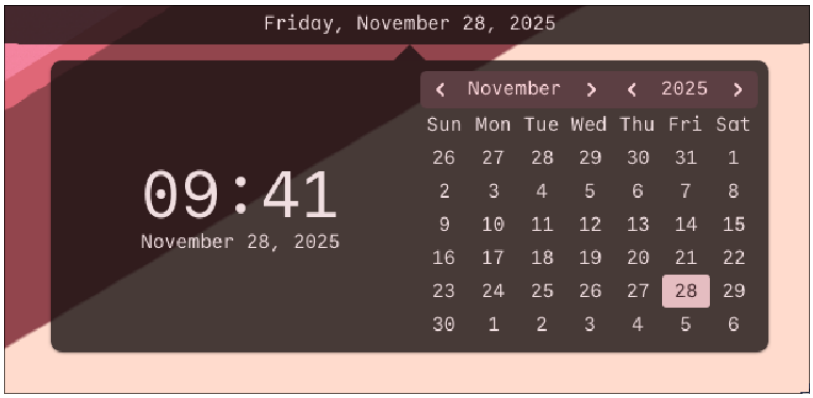


Figura 4.4: *Widget de data e hora*

4.2.4 Menu de início

O menu de início mostra a logo do projeto; botões para suspender, reiniciar, desligar e hibernar o computador e mostra informações do sistema.

Para os botões, foi implementado um serviço básico com um *proxy* para chamar os seguintes métodos da interface "`org.freedesktop.login1`" do *systemd-logind*:

- power_off
- reboot
- suspend
- hibernate

Para pegar as informações do sistema foi utilizado a biblioteca *ffetch* (https://crates.io/crates/ffetch).

4.2.5 Controle de dispositivos *bluetooth*

Para lidar com *bluetooth* foi utilizada a *crate* BlueR (https://crates.io/crates/bluer), que são as *bindings* oficiais do BlueZ. Assim, para controlar os dispositivos *bluetooth*, foi criado um serviço que recebe mensagens de *Input* de tipo:

- `Power(bool)`: Pede para ligar ou desligar o *bluetooth*
- `Devices(canal de resposta)`: Pede lista de dispositivos e envia para o canal de resposta do cliente
- `Connect(DeviceAdress, bool)`: Pede para conectar ou desconectar o dispositivo

E mensagens *Output* de tipo:

Figura 4.5: *Widget do menu de início.*

- `Powered(bool)`: Avisa se o *bluetooth* foi ligado ou desligado
- `Connected(DeviceAddress, bool)`: Avisa se o dispositivo foi conectado ou não

Com isso, foi possível fazer o *widget* da figura 4.6 com a lista de dispositivos *bluetooth* e botões para conectar ou desconectar o dispositivo.

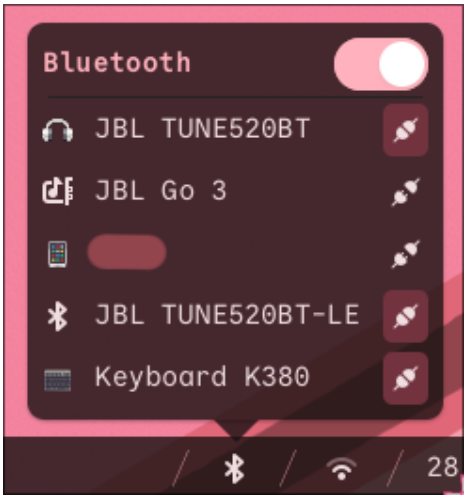


Figura 4.6: *Widget de controle de dispositivos bluetooth*

4.2.6 Controle de conexões de rede

Para controlar as conexões de rede, foi utilizado D-Bus. Mais especificamente, os objetos de “org.freedesktop.NetworkManager”, que, como o nome sugere, depende do programa *NetworkManager*.

O serviço recebe *Inputs* de tipo:

- `Enable(bool)`: liga ou desliga o NetworkManager
- `ListConnections`: lista as conexões
- `DeactivateConnection { ... }`: desativa alguma conexão
- `ActivateConnection { ... }`: ativa alguma conexão

E enviar *Outputs* avisando os clientes sobre mudanças no *NetworkManager*:

- `Connections(Vec<...>)`: Lista de conexões como resposta ao `Input::ListConnections`
- `NetworkingEnabledChanged(bool)`: Diz se o NetworkManager foi ativado ou desativado
- `StateChanged(NMState)`: Avisa sobre mudança no estado na rede (e.g. está conectado, desconectado, prestes a conectar, etc)
- `ConnectivityChanged(NMConnectivityState)`: Avisa sobre mudança de estado da conexão (e.g. se está conectado, se tem acesso à internet, etc.)

Assim, foi possível criar o *widget* da figura 4.7. O botão na barra de tarefas mostra um ícone diferente na barra de tarefas dependendo do estado da conexão. Quando esse botão é clicado, um *popup* é aberto com a lista de conexões conhecidas.

Figura 4.7: *Widget de controle de conexões rede*

4.2.7 Notificações

Para receber notificações, foi necessário criar um *daemon* de notificações D-Bus que se conecta no barramento com o nome de "`org.freedesktop.Notifications`". Essa conexão deve oferecer um objeto "`/org/freedesktop/Notifications`" que implementa a interface "`org.freedesktop.Notifications`". Um exemplo básico de *daemon* de notificações implementado com zbus foi mencionado no exemplo 3.6.

Quando alguma aplicação quer enviar uma notificação, ele encontra a conexão de nome "`org.freedesktop.Notifications`" no barramento e chama o método de `notify()` do objeto "`/org/freedesktop/Notifications`". Quando o *daemon* recebe essa mensagem, ele faz *broadcast* dessa notificação para os *widgets* clientes utilizando a mensagem de *Output*.

Quando o *widget* recebe essa mensagem, ele cria um *widget* para representar essa notificação e adiciona à sua lista de notificações (4.8).

Figura 4.8: *Widget de notificações*

4.2.8 Controle de mídia

Aplicações que tocam vídeo ou algum tipo de mídia utilizam o padrão MPRIS do D-Bus para enviar informações de seus *players* de mídia. Nesse sentido, foi necessário implementar um *proxy* para a interface D-Bus "`org.mpris.MediaPlayer2.Player`" e ficar constantemente ouvindo por novas conexões de *players* MPRIS no barramento.

O serviço envia *Outputs* com o nome do *player* e um evento de tipo:

- Volume(f64)
- Position(i64)
- Metadata(Metadata)
- PlaybackStatus(PlaybackStatus)
- PlaybackRate(PlaybackRate)

Isso permite que o cliente saiba o estado de vários *players* ao mesmo tempo. Analogamente, os *Inputs* também permitem dizer qual *player* deve ser alterado:

- PlayPause(nome do *player*)
- Previous(nome do *player*)
- Next(nome do *player*)

Isso foi suficiente para implementar o *widget* da figura 4.9, com um seletor de *player* e botões para o controle de mídia.

Figura 4.9: *Widget MPRIS*

4.2.9 *Shortcuts*

O *widget* de *shortcut* consiste em um botão que, ao ser clicado, executa alguma aplicação. Isso é feito a aprtir das *desktop entries*,[3] que descrevem o nome, ícone, executável, descrição, entre outras informações de uma aplicação. Não foi necessário criar um serviço, bastou utilizar a estrutura `AppInfo` da biblioteca Gio, incluida no *crate* do GTK.

Figura 4.10: *Widgets de shortcut para Steam, Chromium e Firefox*

A sua configuração também é simples e só precisa do nome da *desktop entry* (4.4).

Programa 4.4 Exemplo de configuração de *shortcut* em formato RON.

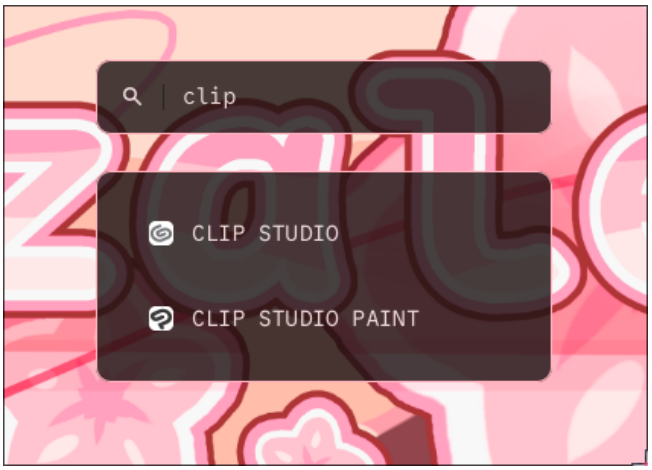
```
1  Shortcut(
2    desktop_entry: "firefox.desktop",
3  )
```

[3] Normalmente, as *desktop entries* de aplicações podem ser encontradas em "*HOME/.local/share/applications*"

4.2.10 Barra de busca de aplicações

De forma semelhante aos *shortcuts*, foi utilizada a estrutura `AppInfo` do Gio para listar todas as aplicações.

Este *widget* tem uma implementação experimental em que o *widget* na barra de tarefas (4.11b) cria, ao ser clicado, a janela com a barra de pesquisa (4.11a). Também seria possível ter implementado isso como uma janela à parte, como o papel de parede e da barra de tarefas.



(a) *Barra de busca de aplicações*

(b) *Botão na barra de tarefas*

Figura 4.11: *Widget de busca de aplicações*

4.2.11 *System tray*

Esta funcionalidade acabou não sendo implementada. Existem vários protocolos diferentes e ainda não há uma especificação de fato padrão:

- *Protocolo de System Tray* (*freedesktop*): específico do X11, utiliza XEMBED e é considerado *legacy* (mas ainda é utilizado)
- *KStatusNotifierItem* (KDE): utiliza D-Bus, não depende do servidor de display
- *StatusNotifierItem* (*freedesktop*): baseado na do KDE, porém ainda está em estado de rascunho
- *Ayatana App Indicator*: baseado no StatusNotifierItem, mas como é recente, não possui adoção ampla no momento

Em teoria, a especificação do *freedesktop* deveria ser o padrão. Porém, na prática, a maioria dos clientes implementa o protocolo do KDE. Nesse sentido, o *System Tray* acabou não sendo implementado.

4.3 *Overview*

A figura 4.12 mostra um *overview* de todos os *widgets*.[4]

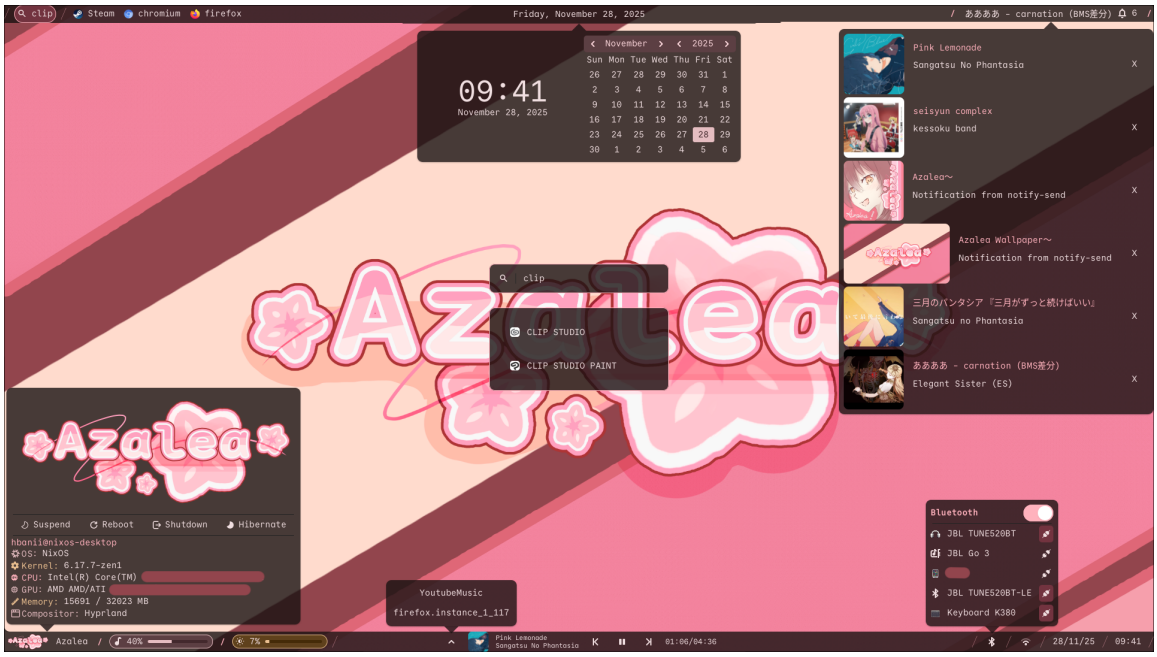


Figura 4.12: *Colagem de fotos com todos os widgets*

[4] Tecnicamente, não é possível mostrar todos os *widgets* ao mesmo tempo, pois o GTK fecha os *popups* quando eles perdem foco. Nesse sentido, a imagem é uma colagem de fotos dos *widgets*.

Capítulo 5

Nix

Nix (https://nixos.org/) é uma linguagem funcional que permite criar ambientes de desenvolvimento, empacotar aplicações, configurar o sistema, etc, de forma declarativa.

5.0.1 Ambiente de desenvolvimento

O ambiente de desenvolvimento, ou *dev shell*, com todas as ferramentas, bibliotecas e LSP, foi configurado utilizando Nix. O exemplo 5.1 mostra como esse shell é usado na prática e 5.2 mostra uma implementação simples de um *dev shell*.

Programa 5.1 Exemplo de como usar um *dev shell* nix.

```
# Inicialmente, rustc não está disponível no PATH
$ rustc --version
bash: line 1: rustc: command not found

# Ao entrar no dev shell, rustc se torna disponível para uso
$ nix develop
$ rustc --version
rustc 1.91.1 (ed61e7d7e 2025-11-07) (built from a source tarball)

# Se sair do dev shell, rustc volta a não estar disponível
$ exit
$ rustc --version
bash: line 1: rustc: command not found
```

5.0.2 Empacotamento

Nix também foi usado para empacotar o programa. Isso permite que ele seja construído pelo comando "`nix build`" sem precisar se procupar em instalar as manualmente bibliotecas no sistema. O exemplo 5.3 mostra como as dependências, *patches*, código-fonte, etc, são declarados.

Programa 5.2 Exemplo de *dev shell* nix.

```nix
# Lista de argumentos
{
  mkShell,
  cargo,
  rustc,
  rustfmt,
  rust-analyzer,
}:

# Cria dev shell
mkShell {
  name = "example-devshell";

  # Variáveis de ambiente
  RUST_SRC_PATH = "${pkgs.rust.packages.stable.rustPlatform.rustLibSrc}";

  # Lista de pacotes que devem ser incluídos no shell
  # compiladores, bibliotecas, formatadores, LSPs, etc
  buildInputs = [
    cargo
    rustc
    rustfmt
    rust-analyzer
  ];
}
```

5.0.3 Testes em máquinas virtuais NixOS

O nix também pode ser usado para criar testes automatizados que criam máquinas virtuais NixOS,[1] tiram uma *screenshot* e salvam os resultados na pasta `result` (5.4).

Idealmente, é desejável testar a aplicação em compositores Wayland que usam bibliotecas diferentes. Nesse sentido, foram escolhidas os seguintes compositores:

- `wlroots`: Sway (5.1), Hyprland (5.2), Wayfire (5.3)
- `smithay`: Niri (5.5)[2]
- `Mir`: Miracle (5.4)

[1] Mais informações sobre testes em máquinas virtuais NixOS, veja a *wiki*: https://wiki.nixos.org/wiki/NixOS_VM_tests

[2] No caso do Niri em particular, foi necessário rodar o teste manualmente com *hardware rendering*, pois Niri não suporta *software rendering*: https://github.com/YaLTeR/niri/issues/218

Figura 5.1: *Azalea rodando no Sway em máquina virtual NixOS.*

Figura 5.2: *Azalea rodando no Hyprland em máquina virtual NixOS.*

Figura 5.3: *Azalea rodando no Wayfire em máquina virtual NixOS.*

Figura 5.4: *Azalea rodando no Miracle em máquina virtual NixOS.*

Programa 5.3 Exemplo simples de como empacotar uma aplicação fantasia em nix.

```
# Lista de argumentos
{
  stdenv,
  fetchurl,
  fetchFromGitHub,

  gtk4
  gtk4-layer-shell
}:

# Cria a "derivação" (pacote)
stdenv.mkDerivation (finalAttrs: {
  pname = "example";
  version = "2.3.4";

  # Código fonte
  src = fetchFromGitHub {
    owner = "enzohideo";
    repo = "some-repository";
    rev = "90bb10e1564c8331b6709957943e95e8cd586769";
    hash = "sha256-WpqZbcKSzCTc9ASB6H6S9urluNE72caBm0x6nc4IGKs=";
  };

  # Aplicar patches
  patches = [
    ./00-fix-something.patch
  ];

  # Bibliotecas
  buildInputs = [
    gtk4
    gtk4-layer-shell
  ];
})
```

Programa 5.4 Exemplo de teste em máquina virtual NixOS

```
testers.runNixOSTest {
  name = "azalea-integration-test";

  # Inicia duas máquinas virtuais com configurações customizadas
  nodes.hyprland = import ./nodes/hyprland.nix;
  nodes.sway = import ./nodes/sway.nix;

  # Configuração padrão utilizadas em todas as máquinas virtuais
  defaults = {
    virtualisation.qemu.options = [ "-vga virtio" ];
  };

  # O script de teste é em Python
  testScript = ''
    def test(machine):
      machine.start()

      machine.wait_for_unit("multi-user.target")
      machine.wait_until_succeeds("pgrep azalea")
      machine.sleep(10)

      with subtest(f"{machine.name}: default"):
        machine.screenshot(f"{machine.name}-default")

      machine.shutdown()

    for machine in machines:
      with subtest(machine.name):
        test(machine)
  '';
}
```

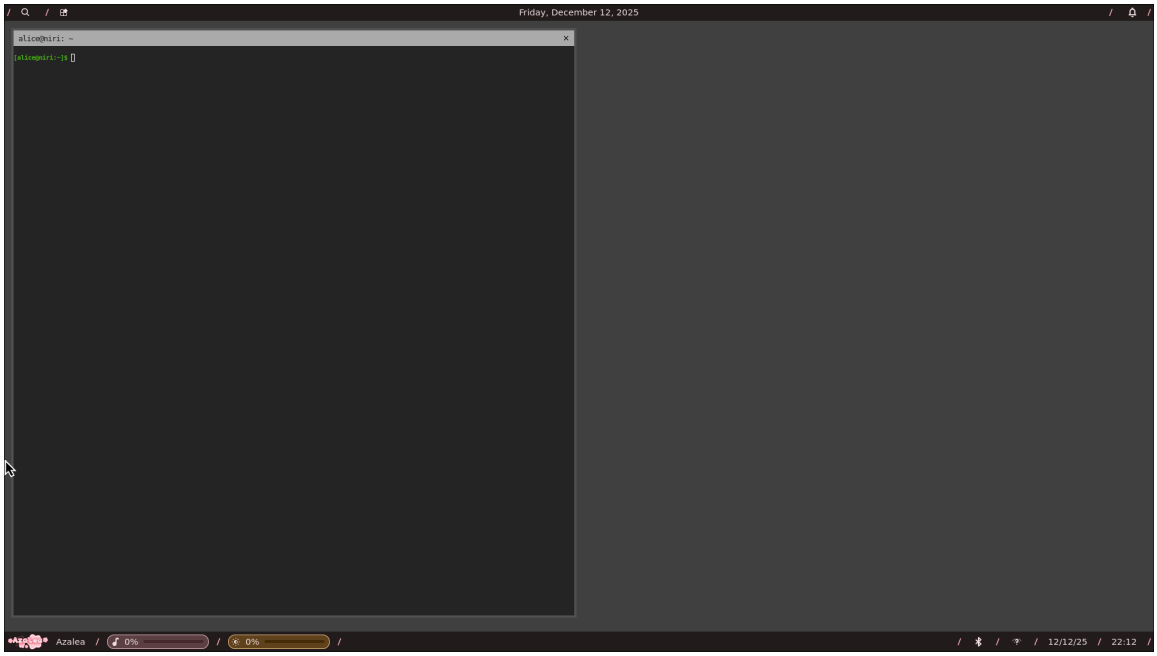


Figura 5.5: *Azalea rodando no Niri em máquina virtual NixOS.*

Capítulo 6

Conclusão

O resultado deste trabalho pode ser encontrado em https://github.com/enzohideo/azalea-desktop-shell. Foram desenvolvidas as seguintes *crates*:

- `azalea`: aplicação principal e re-exportação das bibliotecas internas
- `azalea-core`: cliente-servidor, CLI, estilo Sass, *parsing* dos arquivos de configuração, etc
- `azalea-log`: funções básicas de *logging*
- `azalea-service`: trait do serviço e gerenciador de serviço
- `azalea-derive`: macros *derive*
- `azalea-shell`: implementação dos serviços e *widgets*

A crate *azalea* é responsável por gerar o binário do programa, mas também pode ser importado e usado como biblioteca para implementar outros *shells* gráficos. Sua documentação foi colocada no GitHub Pages do repositório (https://enzohideo.github.io/azalea-desktop-shell/docs/azalea/).

A maioria das funcionalidades esperadas foram implementadas, com exceção do *System Tray*, como comentado na seção 4.2.11. Existem também alguns outros protocolos interessantes que não foram explorados neste projeto, mas que poderiam ser em trabalhos futuros:

- `ext-session-lock-v1`: Protocolo Wayland que permite criar *lockscreens*
- `ext-workspace-v1`: Protocolo Wayland para gerenciar *workspaces* (relativamente recente)
- `greetd-ipc`: Protocolo específico do *greetd* que permite a criação de gerenciadores de login

Por fim, em relação ao objetivo central do projeto de funcionar em múltiplos compositores, pode-se concluir que foi bem-sucedido.

Referências

[Freedesktop 2025a] Freedesktop. *D-Bus*. https://www.freedesktop.org/wiki/Software/dbus/. Acessado em dezembro. 2025 (citado na pg. 5).

[Freedesktop 2025b] Freedesktop. *Wayland*. https://wayland.freedesktop.org/. Acessado em dezembro. 2025 (citado na pg. 3).

[Freedesktop 2025c] Freedesktop. *Wayland Protocols*. https://gitlab.freedesktop.org/wayland/wayland-protocols. Acessado em dezembro. 2025 (citado na pg. 4).

[Relm4 contributors 2025] Relm4 contributors. *GUI development with Relm4*. https://relm4.org/book/stable/introduction.html. Acessado em dezembro. 2025 (citado na pg. 11).

[Steve Klabnik, Carol Nichols, Chris Krycho, Rust community contributors 2025] Steve Klabnik, Carol Nichols, Chris Krycho, Rust community contributors. *The Rust Programming Language, Third Edition*. https://doc.rust-lang.org/stable/book/. Acessado em dezembro. 2025 (citado na pg. 7).

[Wayland Explorer 2025] Wayland Explorer. *wlr layer shell*. https://wayland.app/protocols/wlr-layer-shell-unstable-v1. Acessado em dezembro. 2025 (citado na pg. 4).

[Wikipedia contributors 2025] Wikipedia contributors. *Shell (computing) — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Shell_(computing)\&oldid=1325918852. Acessado em dezembro. 2025 (citado na pg. 3).