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

Modelo de Atores fora de sistemas distribuídos

aplicando o padrão arquitetural em aplicações centralizadas

Jorge Harrisonn Mantovanelli Thomes Vieira

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Paulo Meirelles
Cossupervisor: David Tadokoro

São Paulo
2025

Dedicado a meus avós Elizete e
Jorge e à minha finada mãe Sarah.

Agradecimentos

I've never, ever been this far away from home

[...]

But now I know

It gets dark so I can see the stars

— Sigrid Solbakk Raabe

Agradeço primeiramente aos meus avós por terem cuidado de mim após a minha mãe vir a óbito poucos dias após o meu nascimento. Apesar das dificuldades de crescer vindo do interior de Jaguaré, Espírito Santo, viram o potencial que existia em mim e me incentivaram a seguir com os meus estudos, assim como a minha mãe desejaria.

Agradeço à professora Romilda Sartori que, lá em 2014, enquanto eu estava no quinto ano do ensino fundamental, disse para mim que um dia eu iria me tornar cientista e pediu para que eu me lembrasse dela neste dia. Eu me lembrei.

Agradeço aos professores João Panceri e Weksley Gama do Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo que me fizeram abrir os olhos e sonhar mais alto, visando a Universidade de São Paulo como próximo passo na minha carreira acadêmica.

Agradeço imensamente aos meus amigos Murilo Siloti, Vitor Thompson e Arthur Garcia por terem seguido comigo na difícil jornada que foi o vestibular e também por todos os momentos que compartilhamos juntos no ensino médio.

Agradeço muito aos professores Daniel Batista, Fábio Kon e Paulo Meirelles, assim como ao meu coorientador David Tadokoro, por toda a orientação e suporte que me proporcionaram em projetos que foram além da sala de aula e que me ajudaram a moldar meu futuro profissional.

Por fim, deixo meus agradecimentos aos meus amigos que dividiram essa jornada de 4 anos de estudos e que me ajudaram a chegar até aqui.

Resumo

Jorge Harrisonn Mantovanelli Thomes Vieira. **Modelo de Atores fora de sistemas distribuídos: *aplicando o padrão arquitetural em aplicações centralizadas***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2025.

O Modelo de Atores é uma teoria matemática amplamente utilizada na arquitetura de sistemas distribuídos. Este trabalho explora a aplicação do Modelo de Atores no desenvolvimento de aplicações centralizadas (isto é, não distribuídas), buscando entender se é possível extrair valor deste padrão arquitetural mesmo fora do seu contexto natural. Os objetivos deste trabalho são: propor um arcabouço arquitetural baseado no Modelo de Atores adaptado para aplicações centralizadas em Rust; implementar uma aplicação usando a arquitetura proposta através de um estudo de caso no projeto Patch-Hub; e realizar uma análise qualitativa e quantitativa da aplicação produzida. A aplicação desenvolvida foi avaliada sob os aspectos de testabilidade, desempenho, manutenibilidade e extensibilidade. Os resultados demonstram que, embora o Modelo de Atores não seja naturalmente adequado para aplicações centralizadas, é possível adaptá-lo com sucesso mediante reimaginação tanto do problema quanto do padrão. A arquitetura proposta resultou em melhorias significativas em testabilidade, extensibilidade e isolamento de responsabilidades, embora tenha aumentado a complexidade do código e introduzido sobrecarga em tempo de execução com a comunicação entre atores.

Palavras-chave: Engenharia de Software. Arquitetura de Software. Modelo de Atores. Rust.

Abstract

Jorge Harrisonn Mantovanelli Thomes Vieira. **Actor Model outside distributed systems: *applying the architectural pattern to centralized applications***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

The Actor Model is a mathematical theory widely used in distributed systems architecture. This work explores the application of the Actor Model in the development of centralized applications (i.e., not distributed), seeking to understand whether it is possible to extract value from this architectural pattern even outside its natural context. The objectives of this work are: to propose an architectural framework based on the Actor Model adapted for centralized applications in Rust; to implement an application using the proposed architecture through a case study in the Patch-Hub project; and to perform a qualitative and quantitative analysis of the produced application. The developed application was evaluated from the perspectives of testability, performance, maintainability, and extensibility. The results demonstrate that, although the Actor Model is not naturally suited for centralized applications, it is possible to successfully adapt it through reimagining both the problem and the pattern. The proposed architecture resulted in significant improvements in testability, extensibility, and responsibility isolation, although it increased code complexity and introduced runtime overhead with inter-actor communication.

Keywords: Engenharia de Software. Arquitetura de Software. Modelo de Atores. Rust.

Lista de abreviaturas

API	Interface de Programação de Aplicações (*Application Programming Interface*)
Arc	Contagem de Referência Atômica (*Atomic Reference Counted*)
BEAM	Máquina abstrata de Erlang(*Bogdan's Erlang Abstract Machine*)
DIP	Princípio de Inversão de Dependência (*Dependency Inversion Principle*)
FIFO	Primeiro a Entrar, Primeiro a Sair (*First In First Out*)
HTTP	Protocolo de Transferência de Hipertexto (*Hypertext Transfer Protocol*)
I/O	Entrada e Saída (*Input/Output*)
IME	Instituto de Matemática e Estatística
ISP	Princípio de Segregação de Interface (*Interface Segregation Principle*)
JSON	Notação de Objetos JavaScript (*JavaScript Object Notation*)
LRU	Menos Recentemente Usado (*Least Recently Used*)
LSP	Princípio de Substituição de Liskov (*Liskov Substitution Principle*)
MVC	Modelo-Visão-Controlador (*Model-View-Controller*)
MVVM	Modelo-Visão-Modelo de Visualização (*Model-View-ViewModel*)
mpsc	Múltiplos Produtores, Consumidor Único (*Multiple Producer, Single Consumer*)
OCP	Princípio Aberto/Fechado (*Open/Closed Principle*)
PID	Identificador de Processo (*Process Identifier*)
SOLID	Acrônimo dos cinco princípios: SRP, OCP, LSP, ISP, DIP
SRP	Princípio de Responsabilidade Única (*Single Responsibility Principle*)
TUI	Interface de Usuário de Texto (*Text User Interface*)
UI	Interface de Usuário (*User Interface*)
URL	Localizador Uniforme de Recursos (*Uniform Resource Locator*)
USP	Universidade de São Paulo

Lista de figuras

Lista de programas

Sumário

Capítulo 1

Introdução

A arquitetura de software é um campo de estudo que busca entender como sistemas de software são construídos e organizados para atender aos seus requisitos. Todavia, na prática, projetos costumam dedicar mais atenção às funcionalidades do que aos requisitos de qualidade. Isso traz à tona a necessidade de estudar e entender os padrões arquiteturais existentes e como eles impactam, a longo prazo, a confiabilidade, manutenibilidade, testabilidade, entre outras complicações para um sistema.

Este trabalho tem como objetivo estudar e entender um padrão arquitetural específico chamado de Modelo de Atores. O Modelo de Atores é originalmente uma teoria matemática largamente usada na arquitetura de sistemas distribuídos. Esse modelo enxerga sistemas computacionais em termos de unidades autônomas chamadas de atores que se comunicam entre si através de mensagens. Posteriormente, essa teoria serviu de paradigma base para a criação de linguagens de programação como Erlang e Elixir.

Aqui é feita uma experimentação com o Modelo de Atores tentando aplicá-lo em um contexto distante do qual ele foi originalmente concebido. Isso é feito com a proposição de um arcabouço genérico para a implementação de aplicações centralizadas em Rust. Para validar o arcabouço, foi realizado um estudo de caso aplicando a proposta no projeto PatchHub, uma aplicação de terminal (ou seja, não distribuída) desenvolvida com a linguagem de programação Rust (que não possui o Modelo de Atores como paradigma central). A ideia é entender o esforço necessário para se aplicar um padrão arquitetural fora do seu contexto natural e quais consequências isso traz para o projeto.

Por que sequer seguir adiante com este estudo considerando a premissa que o Modelo de Atores não é apropriado para o domínio desejado? Quando alguém procura um material resistente certamente algodão e água não seriam as primeiras opções a serem pensadas. Todavia, ao se congelar um algodão encharcado, se obtém picrete, um material com resistência comparável ao concreto e que derrete muito mais lentamente que gelo convencional. Isso não é uma garantia de sucesso para combinações pouco usuais, mas significa que pode existir grande potencial a ser explorado.

De modo geral, é constatado que é preciso se depreender um esforço, tanto para se adaptar o arcabouço para um novo domínio, quanto para se reimaginar o problema sendo

resolvido através de uma nova ótica e que a partir disso existe sim valor a ser extraído. Por meio do estudo de caso, pode-se concluir que não é impossível tal adaptação ser bem sucedida e que talvez os padrões arquiteturais sejam mais flexíveis do que se imagina. O sucesso da adaptação está na preservação de uma parte considerável dos benefícios do padrão, mas isso depende da criatividade e do esforço do desenvolvedor.

O restante do texto está estruturado da seguinte forma: No capítulo 2, é feita uma explanação sobre o que é a arquitetura de software e o que são padrões arquiteturais; No capítulo 3, é apresentado o Modelo de Atores do ponto de vista teórico e prático com foco na linguagem de programação Elixir; No capítulo 4, é demonstrada a proposta de um arcabouço genérico para a implementação de aplicações centralizadas usando o Modelo de Atores em Rust com as devidas adaptações; No capítulo 5, é realizado um estudo de caso aplicando a proposta no projeto Patch-Hub; No capítulo 6, são apresentados os resultados da experimentação; No capítulo 7, são apresentadas as conclusões finais e sugestões para futuros trabalhos.

Capítulo 2

Arquitetura de Software

Todo sistema é desenvolvido com um propósito. A forma de saber se o sistema está atendendo ao seu propósito é estabelecendo requisitos que impõem restrições e direcionam o processo de desenvolvimento para a direção correta. Os requisitos em si são uma grande área de pesquisa, mas podem ser principalmente divididos em dois grupos: funcionais e não funcionais.

Requisitos funcionais são restrições que especificam o que o sistema deve ser capaz de fazer, estando fortemente relacionados à lógica de negócio do sistema, ou seja, à sua política de como operar os dados. Um requisito funcional para uma rede social pode ser a capacidade de postar fotos.

Por outro lado, **requisitos não funcionais**, ou requisitos de qualidade, são restrições que definem como algo deve ser feito. Um requisito não funcional para a rede social mencionada pode ser o armazenamento criptografado das fotos dos usuários, impedindo acesso não autorizado a elas.

A **arquitetura de software** é definida como um campo de pesquisa que busca entender como decisões de design afetam a estrutura, comportamento e qualidade geral de um sistema de software, servindo como base para decisões subsequentes. É o campo que lança luz sobre a definição de requisitos de qualidade e como atender a eles.

2.1 Padrões Arquiteturais

Muitas decisões de design estudas pela arquitetura de software podem ser agrupadas de acordo com a categoria de problema que resolvem. Desse modo, se cria um arcabouço de conceitos bem estabelecidos e reutilizáveis chamados de **padrões arquiteturais** (daqui em diante simplesmente referidos como “padrões”).

Padrões tem por objetivo garantir a qualidade do software oferecendo soluções para atender aos requisitos não funcionais de sistemas. Isso significa que podem ser aplicados a qualquer sistema, independentemente de suas funcionalidades, desde que tenham requisitos de qualidade similares.

Todavia, como explorado pelo artigo Kassab *et al.*, 2018, em geral, na indústria, na concepção dos projetos, as funcionalidades são mais levadas em conta do que os requisitos de qualidade. O mesmo artigo ainda afirma que se funcionalidades fossem a única parte importante, não haveria necessidade de se ter uma arquitetura de software.

Por isso, é necessário estudar e entender os padrões existentes, pois eles trazem consequências de longo prazo para confiabilidade, manutenibilidade, testabilidade, entre outras complicações para um sistema.

A seguir, apresentam-se alguns princípios e padrões fundamentais relacionados a arquitetura de software que servirão como base conceitual para as discussões subsequentes neste trabalho.

2.1.1 Princípios SOLID

Os princípios SOLID são um conjunto de diretrizes para o design de software introduzidos por Robert Martin (Martin, 2000). Originalmente, os princípios tinham como alvo principal sistemas orientados a objetos, todavia suas lições podem se extender muito além e ser aplicados a diversos domínios. Seus principais objetivos são criar sistemas mais flexíveis, manuteníveis e extensíveis. Seu nome, é uma sigla com as iniciais de cada um dos 5 princípios, que são:

- ***Single Responsibility*** (SRP): Uma classe deve ter apenas uma razão para mudar, ou seja, deve ter apenas uma responsabilidade.
- ***Open/Closed*** (OCP): Classes devem estar abertas para extensão, mas fechadas para modificação.
- ***Liskov Substitution*** (LSP): Objetos de uma classe devem ser substituíveis por objetos de suas subclasses sem necessidade de modificações.
- ***Interface Segregation*** (ISP): Clientes não devem ser forçados a depender de interfaces que não utilizam.
- ***Dependency Inversion*** (DIP): Classes de alto nível não devem depender de classes de baixo nível. Ambos devem depender de abstrações.

Estes princípios são especialmente relevantes para o contexto do Modelo de Atores que será destrinchado no próximo capítulo. Todavia, já podem-se traçar relações entre esses princípios e as características do Modelo de Atores.

Na definição dos atores que compõem o sistema, cada ator deve, idealmente, ter apenas uma responsabilidade, seguindo, assim, o SRP. A comunicação baseada em mensagens é extremamente poderosa e flexível. Ela permite que os atores sejam facilmente estendidos apenas com a definição de uma nova mensagem e do código que a implementa, seguindo, assim, o OCP.

Além disso, as mensagens permitem a criação de uma abstração para encapsular comportamentos de baixo nível, endereçando, assim, o DIP. Por fim, o ISP pode ser garantido ao corretamente segmentar os atores de acordo com as dependências do sistema,

permitindo, assim, que cada ator dependa diretamente apenas de atores relevantes para a sua responsabilidade.

2.1.2 Padrão MVC

O padrão Model-View-Controller (MVC) (Reenskaug, 1979) é um padrão arquitetural que divide uma aplicação em três componentes interconectados, cada um com responsabilidades bem definidas. Esse padrão surgiu para resolver problemas de acoplamento entre a lógica de negócio e a apresentação, permitindo que essas camadas evoluam de forma independente.

Os três componentes do padrão MVC são:

- **Model**: Representa os dados e a lógica de negócio da aplicação. É responsável por manter o estado do sistema e realizar operações sobre os dados. O Model não possui conhecimento sobre como os dados são exibidos nem como são coletados.
- **View**: É responsável pela apresentação dos dados ao usuário. A View consome dados do Model e os exibe de forma adequada à interface (web, desktop, terminal, etc.). A View não deve conter lógica complexa, apenas a lógica de apresentação.
- **Controller**: Funciona como intermediário entre o usuário e o Model. Recebe entradas do usuário, processa essas entradas e atualiza o estado da aplicação no Model.

O MVC oferece benefícios significativos para a manutenibilidade e testabilidade de sistemas. A separação de responsabilidades permite que a lógica de negócio seja testada independentemente das interfaces humano-computador, facilitando testes unitários. Além disso, múltiplas Views e múltiplos Controllers podem ser criados para interagir com o mesmo Model, permitindo reutilização de código e adaptação a diferentes contextos.

No entanto, o padrão MVC tradicional apresenta algumas limitações quando aplicado na prática. Em muitas de suas implementações, a View e o Controller acabam sendo implementados de forma muito acoplada, o que acabou por dar origem a novas derivações do padrão, como o Model-View-ViewModel (MVVM) (Kouraklis, 2016).

2.2 Avaliação de Adequação de Padrões

O primeiro e mais importante fator a ser analisado para se identificar a adequação de um padrão para um projeto é, evidentemente, o casamento entre os requisitos de qualidade que o padrão aborda e os requisitos de qualidade do sistema.

Uma outra característica a ser considerada é o contexto da aplicação. Existem padrões que abordam desafios comuns para aplicações de nuvem, outros para aplicações de interface gráfica. Escolher um padrão de um domínio distinto irá fornecer poucas ferramentas úteis para a resolução do problema.

Por fim, é necessário entender os custos que aquele padrão traz. Qual impacto esperado na *performance* do sistema? O quanto ele enrijece a estrutura do projeto? O quão mais complicada fica a organização do código? Todas essas perguntas precisam ser feitas durante a análise.

2.2.1 Adequação do Modelo de Atores

O Modelo de Atores será bem definido do ponto de vista teórico e prático no próximo capítulo. Porém, avaliando o Modelo de Atores em termos de adequação para a criação de aplicações centralizadas, é possível identificar que o padrão não é adequado para o contexto.

Primeiramente, pelo fato de que o Modelo de Atores foi originalmente concebido para sistemas distribuídos, isso por si só já é um indicativo de que ele não é adequado para o contexto de aplicações centralizadas. Aplicações distribuídas têm necessidade de ser resilientes, escaláveis, tolerantes a falhas e permitir comunicação assíncrona remota, nenhuma dessas características é algo fortemente procurado em aplicações centralizadas.

Ademais, a linguagem de programação Rust, escolhida para a implementação do projeto, não possui o Modelo de Atores como paradigma central, diferentemente de Elixir. Apesar de isso não ser um impeditivo para a aplicação do padrão, é uma característica que deve ser considerada. Rust é uma linguagem que prioriza a segurança de memória e a concorrência confiável através de checagens em tempo de compilação, enquanto o Modelo de Atores é tradicionalmente implementado em tecnologias mais flexíveis e dinâmicas.

Desse modo, com os critérios de adequação utilizados, é possível concluir que o Modelo de Atores não é adequado para a criação de aplicações centralizadas em Rust e que aplicar ele neste contexto sem nenhum cuidado especial irá trazer consequências negativas para o projeto.

O objetivo deste trabalho é então realizar um esforço para adaptar o Modelo de Atores para o contexto de aplicações centralizadas em Rust, buscando entender se é possível extrair valor dele mesmo fora do seu contexto natural. Esse experimento vai permitir:

- Explorar os limites do Modelo de Atores
- Avaliar se os benefícios do padrão se mantêm em aplicações centralizadas
- Identificar possíveis adaptações do padrão
- Contribuir para o conhecimento sobre a aplicabilidade de padrões arquiteturais

Capítulo 3

Modelo de Atores

O Modelo de Atores de computação foi estabelecido por Carl Hewitt em 1973, baseado em trabalhos anteriores de Peter Bishop e Richard Steiger (Hewitt, 2010). Seus objetivos eram abordar os novos desafios impostos pelos sistemas massivamente concorrentes e paralelos que emergiram com a computação em nuvem e arquiteturas many-core.

Seus objetivos foram alcançados criando um modelo computacional que é inerentemente concorrente. Uma grande mudança dos modelos anteriores é que ele é baseado em leis físicas, ao invés de álgebra pura. A hipótese mais importante defendida pelo trabalho de Hewitt é:

> **Hipótese:** *Toda computação fisicamente possível pode ser diretamente implementada com Atores*

O Modelo de Atores representa uma reformulação completa da computação concorrente, passando de máquinas de estado globais para sistemas distribuídos, assíncronos e baseados em passagem de mensagens que melhor refletem a realidade física dos sistemas de computação modernos.

3.1 O que é um Ator?

Hewitt propõe este primitivo chamado Ator que é identificado por um endereço. Atores são entidades que se comunicarão entre si através de mensagens. Uma vez que uma mensagem é recebida, um Ator pode:

- Enviar mensagens para Atores cujos endereços conhece
- Criar novos Atores
- Modificar seu próprio comportamento para lidar com mensagens futuras recebidas

Mensagens são a unidade de comunicação e são passadas de forma assíncrona entre atores. De acordo com o princípio de segurança e localidade, ao processar uma mensagem recebida, um Ator pode apenas enviar mensagens para Atores cujos endereços foram:

- Conhecidos anteriormente
- Parte da mensagem recebida
- Criados durante o processamento atual

3.2 Indeterminação

Aqui vem um dos princípios fundamentais do Modelo de Atores e o que o diferencia de outros padrões. A ordem de chegada das mensagens não é garantida e o Modelo de Atores não tentará resolver isso, mas sim tratar isso como uma propriedade natural do domínio. O Modelo de Atores é projetado para sistemas distribuídos, o que significa que dois atores podem estar se comunicando através de longas distâncias físicas.

Isso significa que as mensagens enviadas podem levar tempo ilimitado para chegar na máquina destino, e mesmo que cheguem exatamente ao mesmo tempo, os árbitros de hardware que as ordenarão não garantirão nenhum tipo de ordenação. Considerar os fatores físicos é o que diferencia o Modelo de Atores de padrões de sistemas concorrentes puramente algébricos.

A ordenação indeterminada de mensagens, somada ao fato de que as mensagens podem alterar o comportamento de um Ator, tem por consequência que as mesmas condições iniciais podem resultar em resultados diferentes. Além disso, o próprio sistema é dito não ter um estado global definido. Como a comunicação é assíncrona, a qualquer momento é possível ter mensagens transito ou Atores em estado transitório não sendo possível garantir que em algum momento todos os atores estarão em estados bem definidos.

Em resumo, a indeterminação é uma característica fundamental do Modelo de Atores que é essencial para a garantia de justiça e deve ser sempre considerada durante o desenvolvimento de um sistema para assegurar o seu correto funcionamento.

3.3 Implementações Conhecidas

O Modelo de Atores recebeu a devida atenção e saiu do estado de um modelo/teoria computacional para se transformar em um padrão arquitetural com foco em abordar sistemas distribuídos. Bibliotecas e arcabouços como Akka (Java) e Orleans (C#) fornecem suporte ao Modelo de Atores para linguagens de programação. No entanto, a implementação mais proeminente e bem-sucedida é a linguagem de programação Elixir.

Na década de 1980, a empresa de telecomunicações sueca Ericsson experimentou com linguagens de programação funcional para abordar desafios de telecomunicações, desenvolvendo eventualmente Erlang. Erlang focou em concorrência, distribuição e tolerância a falhas implementando o Modelo de Atores no nível da linguagem. BEAM, a máquina abstrata que serviu como *runtime* para programas Erlang, representou a maior conquista da Ericsson neste domínio. (Na época, o termo "máquina virtual" ainda não era aplicado a tais sistemas.) No entanto, a sintaxe do Erlang era considerada complexa, criando uma barreira para adoção mais ampla.

No início dos anos 2010, José Valim estava desenvolvendo soluções para sistemas distribuídos, os mesmos desafios que a Ericsson havia abordado três décadas antes. Ao descobrir Erlang e BEAM, ele reconheceu sua eficácia, mas os achou difíceis de usar. Isso levou à criação de Elixir, que combinou o poder do BEAM e a flexibilidade do Modelo de Atores com uma sintaxe mais acessível inspirada em Ruby.

Ao contrário de implementações baseadas em bibliotecas que adicionam suporte ao Modelo de Atores a linguagens existentes, Elixir torna o Modelo de Atores o paradigma central da linguagem. Isso é alcançado através de suporte nativo para *green threads*, tolerância a falhas e comunicação inter-processo remota. Esses são diferenciadores que estabeleceram Elixir como a principal plataforma do Modelo de Atores.

Esta linguagem dinâmica continua ganhando reconhecimento, sendo eleita como a segunda linguagem de programação mais admirada por três anos consecutivos (2022 a 2024) e sendo a terceira linguagem mais admirada em 2025 na Pesquisa de Desenvolvedores do Stack Overflow.[1] Portanto, este trabalho usará Elixir como a implementação de referência e exemplo de estado da arte do Modelo de Atores.

3.3.1 A Linguagem de Programação Elixir

A filosofia de design do Elixir difere fundamentalmente das linguagens orientadas a objetos. Ao invés de classes instanciadas em objetos, Elixir usa módulos que geram atores por design. Esta abordagem torna o Modelo de Atores o paradigma central da linguagem.

Os atores no Elixir são representados por processos Erlang (daqui em diante referidos simplesmente como processos). Estes processos leves são criados usando a função `spawn`, que recebe três parâmetros: um módulo, um nome de função daquele módulo e uma lista de argumentos. Quando chamada, `spawn` executa a função especificada em um processo dedicado a ele e retorna um *Process Identifier* (PID) que serve como o endereço do ator.

A comunicação entre atores acontece através de passagem de mensagens usando duas funções: `send` para despachar mensagens para um processo usando seu PID como referência, e `receive` para lidar com mensagens recebidas dentro de um processo. Este mecanismo de passagem de mensagens assíncrono forma a base das capacidades concorrentes e distribuídas do Elixir.

Como Elixir é construída extensivamente sobre o ecossistema do Erlang e executa na máquina virtual BEAM, referências tanto ao Erlang quanto ao BEAM aparecerão frequentemente ao longo desta discussão.

O Modelo de Atores em Elixir

A implementação do Modelo de Atores em Elixir é simples, requerendo esforço mínimo para criar sistemas de atores concorrentes e distribuídos. O design da linguagem torna a programação baseada em atores natural ao invés de um padrão adicional a aprender.

[1] Dados disponíveis em: StackOverflow Developer Survey 2025, StackOverflow Developer Survey 2024, StackOverflow Developer Survey 2023 e StackOverflow Developer Survey 2022.

Mensagens em Elixir são representadas por qualquer estrutura de dados válida: átomos, strings, números, tuplas, listas, ou mesmo estruturas aninhadas complexas. Esta flexibilidade permite que desenvolvedores projetem protocolos de mensagem que melhor se adequem às necessidades de sua aplicação sem serem restringidos por formatos de mensagem rígidos.

Endereços são *Process Identifiers* do Erlang (PIDs), que são identificadores únicos automaticamente gerados pela máquina virtual BEAM quando um processo é gerado. Estes PIDs servem como o endereço do ator, permitindo roteamento de mensagens tanto em ambientes locais quanto distribuídos.

Manipulação de mensagens ocorre através da função `receive`, que permite que um processo aguarde o recebimento de uma mensagem, faça casamento de padrões. Este mecanismo permite que atores processem diferentes tipos de mensagens com lógica de manipulação apropriada, tornando o sistema tanto robusto quanto expressivo.

Noções Básicas do Elixir

Elixir é uma linguagem de programação dinamicamente tipada, o que significa que tipos são verificados apenas em tempo de execução. Versões mais novas da linguagem estão experimentando com anotações de tipo estáticas opcionais, mas ainda em estágios iniciais de desenvolvimento e adoção.

Existem muitos tipos de dados nativos diferentes no Elixir que podem ser vistos no programa 3.1.

Programa 3.1 Tipos de Dados em Elixir.

```
1    i = 123
2    f = 3.14
3    s = "héllo" # UTF-8
4    c = 'hello' # Charlists (lista de ints)
5    b = true
6    a = :atom
```

A sintaxe especial `:identificador` mostrada no exemplo 3.2 é usada para declarar átomos, um conceito do Erlang. Eles são constantes nomeadas por si mesmas e usadas em muitos cenários. Eles são usados para representar códigos de status, nomes de funções, tags ou mesmo representar valores de dados primitivos.

Programa 3.2 Átomos em Elixir.

```
1    true == :true
2    false == :false
3    nil == :nil
```

Quando se trata de dados compostos, Elixir tem 2 tipos principais (para o escopo deste trabalho): tuplas e listas. As primeiras têm tamanho fixo e são rapidamente indexáveis. As últimas são listas encadeadas simples com comprimento dinâmico. Tuplas são muito

usadas para representar dados estruturados simples. Os dados como um todo no Elixir são imutáveis, o que significa que você não pode alterar dados, mas apenas usá-los para produzir novos dados. Veja o exemplo 3.3.

Programa 3.3 Estruturas de Dados em Elixir.

```
response = {:ok, "Some message"}
status = elem(response, 0) # :ok
response = put_elem(response, 1, "Another message") # {:ok, "Another message
    "}

data = [1, 2, 3, 4]
data = ["0" | data] # ["0", 1, 2, 3, 4]
data = data ++ [5] # ["0", 1, 2, 3, 4, 5]
```

O = é chamado de operador de *binding*, isso porque não simplesmente faz atribuição, mas pode ser usado para casamento de padrões. Veja o exemplo 3.4.

Programa 3.4 Casamento de Padrões em Elixir.

```
x = 1
1 = x # Isso funciona
2 = x # Isso irá causar um erro

# Desestruturar tuplas
{status, message} = {:err, "the system panicked"}

IO.puts "It's a #{status}. Because #{message}"
# It's a err. Because the system panicked
```

Mais amplamente, correspondência de padrões é demonstrada no exemplo 3.5 e pode ser feita com a sintaxe `->`.

Programa 3.5 Estruturas de Controle em Elixir.

```
response = {:ok, "Content"}

case response do
  {:ok, content} -> IO.puts "Success with content: #{content}"
  {:err, cause} -> IO.puts "Ugh, it failed due to: #{cause}"
end
```

Em Elixir, existem algumas formas diferentes de declarar funções, e elas podem ser agrupadas com o uso de módulos. Veja o exemplo 3.6 de um módulo com funções.

3.3.2 Exemplo em Elixir

No exemplo 3.7, é definido um ator echo simples. Ele recebe uma mensagem que deve conter o endereço do remetente e alguma mensagem. Ele então envia de volta ao remetente uma mensagem com conteúdo `:ok` e a mensagem recebida prefixada com `Echoing:` .

Programa 3.6 Módulos e Funções em Elixir.

```elixir
defmodule Math do
  # Função pública
  def add(a, b) do
    a + b
  end

  # Forma abreviada de uma linha
  def mul(a, b), do: a * b

  # Função privada (apenas chamável dentro do módulo)
  defp square(x), do: x * x

  # Múltiplas declarações de função com casamento de padrões
  def abs(n) when is_number(n) and n < 0, do: -n
  def abs(n) when is_number(n), do: n
end

IO.puts(Math.add(2, 3)) # 5
IO.puts(Math.mul(4, 5)) # 20
```

Programa 3.7 Exemplo de Ator Echo em Elixir.

```elixir
defmodule Echo do
   def init do
      # Aguarda por uma mensagem
      receive do
         # Obtém o remetente e conteúdo da mensagem
         {sender, msg} ->
            # Ecoa a resposta com sucesso (`:ok`)
            send(sender, {:ok, "Echoing: #{msg}"})
            # Loop
            init()
      end
   end
end

defmodule Example do
   def main do
      # Inicia o ator
      addr = spawn(Echo, :init, [])
      # Envia uma mensagem
      send(addr, {self(), "Hello world"})

      # Aguarda a resposta
      receive do
         {:ok, response} ->
            # Exibe a resposta
            IO.puts(response)
      end
   end
end
```

Finalmente, há uma chamada recursiva para `init` para que o ator possa lidar com mais mensagens, já que Elixir não tem loops.

No módulo `Example`, a chamada para `spawn` iniciará o ator através da função `init` do módulo `Echo` sem parâmetros. Esta função retornará o PID do processo criado que serve como o endereço do ator. A função `send` envia ao ator recém-criado uma tupla onde o primeiro elemento é o PID do processo atual e o segundo é a string `"Hello world"`.

Neste momento, a chamada para `receive` na função `init` entra em ação. Ela receberá a mensagem e retornará ao remetente uma nova tupla onde o primeiro elemento é `:ok` e o segundo é a string `"Echoing: Hello world"`. Por causa da chamada recursiva, este ator está pronto para receber mais mensagens.

A chamada para `receive` na função `main` agora lidará com a resposta do ator. Ela aguardou pacientemente por uma resposta e pode agora verificar com casamento de padrões que a resposta realmente contém um `:ok` e exibir a string `Echoing: Hello world` no terminal.

Note que esse código não é executado sequencialmente. A chamada de `send` do lado do módulo `Example` não vai aguardar o ator de echo processar a mensagem e gerar uma resposta. Assim que a mensagem é enviada o módulo `Example` fica parado na chamada `receive` aguardando uma resposta.

Capítulo 4

Proposta de Modelos de Atores em Rust

Uma das principais contribuições deste trabalho é a proposta de uma estratégia para aplicações centralizadas escritas em Rust usando o Modelo de Atores. Uma vez que a teoria sobre o Modelo de Atores e as implementações conhecidas foram discutidas, é hora de adaptar suas abstrações para as necessidades e características específicas de aplicações centralizadas em Rust.

4.1 Rust

O Modelo de Atores é amplamente conhecido por sua dinamicidade e flexibilidade, com a maioria de suas implementações famosas sendo em *runtimes* flexíveis como BEAM ou JVM. Rust (Rust Project Developers, 2025) é bem o oposto disso, uma linguagem moderna com um sistema de tipos forte e um compilador poderoso o suficiente para validar quase tudo em tempo de compilação.

Pode soar contraditório usar uma linguagem rígida com um padrão flexível, mas é exatamente o propósito deste trabalho: experimentar com o Modelo de Atores fora do seu domínio específico. Assim, além de testar no contexto centralizado, iremos implementá-lo numa tecnologia que não foi pensada pra isso.

Apesar desta distância, Rust ainda possui diversos recursos de linguagem análogos a mecanismos fornecidos por Elixir, tornando essa adaptação possível. Além disso, se Elixir foi eleita a segunda linguagem mais amada nos últimos 3 anos, a mesma pesquisa aponta Rust como sendo a linguagem mais amada desde 2016. Vale ressaltar que Rust 1.0 foi lançado em 2015.

Rust é conhecido por seus mecanismos de segurança que forçam verificação em tempo de compilação, tornando diversos estados inválidos impossíveis de serem representados como código. A seguir veremos alguns dos conceitos mais importantes aqui.

4.1.1 Segurança de Memória

Um espaço de memória tem um, e apenas um, proprietário (uma variável), este proprietário determina o ciclo de vida do espaço de memória. Uma vez que o proprietário é destruído, o espaço de memória é liberado. O proprietário também determina se a região é mutável ou não.

Programa 4.1 Declaração de variáveis e mutabilidade.

```
1    fn main() {
2       let a = 10;
3       let mut b = 20;
4
5       // a += 1; // Proibido: a não émutável
6       b += 1; // Permitido: b émutável
7
8       // Quando o código termina aqui, a e b são liberados da memória
9    }
```

Programa 4.2 Transferência de propriedade.

```
1    fn main() {
2       let data = [1, 2, 3, 4]; // Cria variável imutável
3       // As duas operações a seguir são proibidas pelo compilador
4       // data = [2, 3, 4, 5];
5       // data[0] = 0;
6
7       let mut data2 = data; // Variável mutável data2
8       // Uso de data proibido pelo compilador desse ponto em diante,
9       // já que seu conteúdo foi transferido para data2
10      // println!("{}", data[0]); // Erro de compilação
11
12      // Permitido
13      data2[0] = 0;
14      data2 = [2, 3, 4, 5];
15   }
```

Um valor pode ser emprestado com o uso de referências, mas existem regras checadas pelo compilador:

1. Referências não podem ser nulas
2. As referências não podem existir por mais tempo do que o dono original
3. Podem existir infinitas referências imutáveis (`&`) a um valor
4. Pode existir no máximo uma referência mutável (`&mut`) a um valor (desde que ele seja mutável)
5. Um valor não pode ter ambas referências ao mesmo tempo

Programa 4.3 Referências imutáveis.

```rust
fn main() {
    let numbers = vec![1, 2, 3, 4];
    let nref = &numbers;
    println!("{}", nref[1]); // -> 2

    // Permitido: múltiplas referências imutáveis
    let nref2 = &numbers;

    // Proibido: não pode ter referência mutável
    // quando já existem referências imutáveis
    // let mref = &mut numbers;
}
```

Programa 4.4 Referências mutáveis.

```rust
fn main() {
    let mut numbers = vec![1, 2, 3];
    let nref = &mut numbers;

    // Permitido: operação mutável usando &mut
    nref.push(4);

    // Proibido: só pode haver uma referência mutável
    // let nref2 = &mut numbers;

    // Proibido: não pode usar o valor original
    // enquanto existe referência mutável
    // numbers.push(5);
}
```

Programa 4.5 Ciclo de vida de referências.

```rust
fn main() {
    let numbers = vec![1, 2, 3, 4];
    let nref = &numbers;
    println!("{}", nref[1]); // -> 2

    drop(numbers); // Libera numbers da memória

    // A partir deste ponto, o compilador proibirá
    // o uso de tanto numbers quanto nref
    // println!("{}", nref[1]); // Erro de compilação
}
```

4.1.2 Sistema de Tipos

O sistema de tipos forte e expressivo é uma característica-chave do Rust. Ele não tem herança, mas compensa isso com tuplas, structs, enums e traits.

Tuplas

Uma tupla é simplesmente um tipo produto, tem tamanho fixo e cada campo (nomeado por seu índice) pode ter um tipo diferente. Não é uma coleção!

Programa 4.6 Declaração e uso de tuplas.

```
1   fn main() {
2       let value: (i32, f32) = (20, 3.14);
3       println!("{}", value.0); // -> 20
4   }
```

Structs

Um struct é similar a uma tupla, mas tem campos nomeados e pode ter métodos com o uso de um bloco `impl`:

Programa 4.7 Definição e uso de structs.

```
1   struct Point {
2       timestamp: i32,
3       value: f32
4   }
5
6   impl Point {
7       // Método que pode mutar o valor de self
8       fn add(&mut self, value: f32) {
9           self.value += value;
10      }
11
12      // Método estático que serve como construtor
13      fn new(timestamp: i32, value: f32) -> Self {
14          Self { timestamp, value } // Sintaxe abreviada
15      }
16  }
17
18  fn main() {
19      let mut p = Point::new(20, 2.14);
20      p.add(1.0);
21  }
```

Existe também um construto chamado tuple-struct que é um struct mas os campos são nomeados por seus índices, como uma tupla:

Programa 4.8 Tuple-struct.

```
1   struct Coordinate(i32, i32);
2
3   fn main() {
4       let origin = Coordinate(0, 0);
5   }
```

Enums

Diferente da maioria das linguagens, enums em Rust podem ter payloads já que cada variante pode ser tratada como um struct:

Programa 4.9 Enums com payloads.

```
1   enum Coordinate {
2       Polar {
3           theta: f32,
4           radius: f32
5       },
6       Cartesian(f32, f32),
7       Origin,
8   }
9
10  fn main() {
11      let origin: Coordinate = Coordinate::Origin;
12      let point1: Coordinate = Coordinate::Cartesian(1.0, 1.0);
13      let point2: Coordinate = Coordinate::Polar { theta: 45.0, radius: 2.0 };
14  }
```

Traits

Traits são o equivalente Rust de uma interface, define métodos que devem ser implementados por um tipo para que ele tenha uma dada característica. São úteis para limitar tipos genéricos ou sobrecarregar operadores. Veja um exemplo no programa 4.10.

Com conhecimento básico da linguagem, é momento de começar a introduzir os elementos da proposta em si. Para isso, os elementos da linguagem Elixir que foram discutidos previamente serão usados como referência fortemente.

4.2 Concorrência e Paralelismo

Esta proposta contempla o uso de *green-threads* que não têm suporte nativo em Rust. Isso é porque a linguagem, ao contrário de Elixir, Golang, ou Java, tem um *runtime* muito mínimo, similar ao *runtime* do C. O suporte para *green-threads* é fornecido pela biblioteca Tokio (Tokio Contributors, 2025) e eles são chamados de *tasks* e serão usados de forma equivalente aos processos Erlang.

As tasks são criadas e executadas com isolamento, seja de forma concorrente ou paralela, dependendo da quantidade de threads do sistema operacional que estão sendo utilizadas

Programa 4.10 Implementação de traits.

```
use std::fmt::{Display, Formatter, Error};

// Display éum trait usado para formatação de string com {}
impl Display for Coordinate {
   fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error> {
      match self {
         Coordinate::Origin => write!(f, "Origin"),
         Coordinate::Cartesian(x, y) => write!(f, "({}, {})", x, y),
         Coordinate::Polar { theta, radius } =>
            write!(f, "{} ({})", radius, theta)
      }
   }
}

fn main() {
   let coord = Coordinate::Polar { radius: 10.0, theta: 75.0 };
   println!("A coordenada é{}", coord); // -> 10 (75)
}
```

pelo ambiente de execução Tokio. Isso permite que múltiplas tasks sejam executadas simultaneamente sem a necessidade de gerenciar manualmente threads do sistema operacional.

Programa 4.11 Exemplo de tasks concorrentes com Tokio.

```
use tokio;

#[tokio::main]
async fn main() {
   // Gera a primeira task
   let task1 = tokio::spawn(async {
      println!("Task 1 executando!");
   });

   // Gera a segunda task
   let task2 = tokio::spawn(async {
      println!("Task 2 executando!");
   });

   // Aguarda ambas as tasks terminarem
   let _ = tokio::join!(task1, task2);
}
```

No programa 4.11, duas tasks são criadas usando `tokio::spawn`, cada uma executando de forma independente e concorrente (sem garantia de ordenação). A função `tokio::join!` aguarda que ambas as tasks sejam concluídas antes de prosseguir.

Similarmente ao que foi apresentado nos exemplos com Elixir, onde um ator teria um processo dedicado para lidar com a lógica de processamento de mensagens, aqui será feito uso de *tasks* para isso.

4.3 Compartilhamento de Memória

O isolamento de memória é desejável e uma característica nativa do Modelo de Atores, já que atores são projetados para executar em ambientes físicamente isolados. Isso aumenta a segurança, mas pode degradar a *performance* já que o compartilhamento de dados entre máquinas remotas só pode ser feito por cópia.

Mas esse não é o caso para esta proposta, já que a aplicação não é distribuída, temos a garantia que a memória sempre pode ser compartilhada. Além disso, os problemas com compartilhamento de memória em sistemas concorrentes estão geralmente relacionados a condições de corrida e mutabilidade indesejada.

Assim como Elixir garante a imutabilidade dos valores para impedir condições de corrida na comunicação entre processos, Rust e Tokio fornece diversos mecanismos nativos eficientes para compartilhar espaços de memória de forma segura entre *tasks*. Desse modo, usando a abstração correta no cenário correto, é possível de se obter resultados interessantes.

O tipo `Arc<T>` será o principal utilizado para dados complexos, já que permite que um espaço de memória imutável seja compartilhado com contagem de referência atômica (uma vez que ninguém está referenciando-o, ele é liberado). Isso garante que múltiplas tasks possam acessar os mesmos dados de forma segura sem a necessidade de cópia.

4.4 Passagem de Mensagens e Endereços

Tanto Rust quanto Tokio têm uma abstração nativa de passagem de mensagens chamada de canal. Ao contrário da função `send` do Elixir que recebe como parâmetros um endereço e um dado qualquer como mensagem, o método `send` dos canais é fortemente tipado quanto à mensagem e usa um endereço bem definido.

Existem alguns tipos diferentes de canais. Os 2 principais tipos para o escopo deste trabalho são: *mpsc* e *oneshot*.

4.4.1 `mpsc` - *Multiple producer, single consumer*

Um canal de múltiplos produtores, consumidor único é a abstração que será usada como endereço de ator. Ele tem 2 extremidades: `Sender` e `Receiver`. O `Sender` pode ser compartilhado entre todos os atores que querem se comunicar com um ator particular apenas criando uma duplicata com o método `clone`. A extremidade `Receiver` é única e será mantida na *task* onde a lógica do ator se encontra e receberá mensagens até o canal ser fechado, ou seja, até não existirem mais clones da extremidade `Sender` correspondente.

Note que o princípio de localidade e segurança do Modelo de Atores nos garante que ninguém deve ser capaz de descobrir o endereço de um ator sem que esse tenha lhe sido dado. Portanto, no momento que não existem mais extremidades `Sender` a um ator, este garantidamente não receberá mais nenhuma mensagem e pode ser finalizado. Isso acontece de forma natural em Rust com o uso de um *loop*, enquanto que em Elixir um ator não recebe nenhum indicativo de que seu endereço se tornou desconhecido.

Programa 4.12 Exemplo de loop de recebimento de mensagens.

```
1    // Loop que processa mensagens até o canal ser fechado
2    while let Some(message) = rx.recv().await {
3       match message {
4          Message::SetThing { value } => {
5             println!("Definindo valor: {}", value);
6          }
7          Message::GetThing { tx } => {
8             let _ = tx.send(42); // Envia resposta
9          }
10      }
11   }
12   // Quando não há mais Senders, rx.recv() retorna None
13   // e o loop se encerra automaticamente
14   println!("Ator finalizando - canal fechado");
```

No programa 4.12, o *loop* `while let Some(message) = rx.recv().await` continuará processando mensagens enquanto houver extremidades `Sender` ativas. Quando todas as extremidades `Sender` são descartadas (saem de escopo ou são explicitamente liberadas com o uso de `drop`), o método `recv()` retorna `None`, fazendo com que o *loop* se encerre naturalmente e a *task* do ator seja finalizada.

Programa 4.13 Exemplo equivalente em Elixir.

```
1    def loop do
2       receive do
3          {:set_thing, value} ->
4          IO.puts("Definindo valor: #{value}")
5          loop()
6          {:get_thing, sender} ->
7          send(sender, {:ok, 42})
8          loop()
9       end
10   end
```

O programa 4.13 em Elixir é um ator que pode se tornar um ator zumbi, ou seja, que está consumindo recursos ainda que não seja mais usado. Para resolver isso é necessário o uso de um *timeout* ou de uma mensagem que indique ao ator que ele pode terminar.

É importante notar que o Rust garante, em tempo de compilação, que o receptor de um canal é único. Assim, é garantido que as mensagens chegam ao destino correto. Além disso, o canal tem uma fila com tamanho configurável, e é garantido que a mensagem chegará se a fila não estiver cheia.

4.4.2 oneshot

Um canal que é destruído após uma mensagem ser enviada. Como os canais são unidirecionais, se uma mensagem enviada precisar de uma resposta, um canal oneshot deverá ser criado para envio da resposta.

Também vale ressaltar que o "uso após liberação"de um canal oneshot é verificado em tempo de compilação.

4.5 Representação do Ator

Uma vez definidas quais abstrações a serem usadas, é hora de definir como um ator será em termos de código Rust.

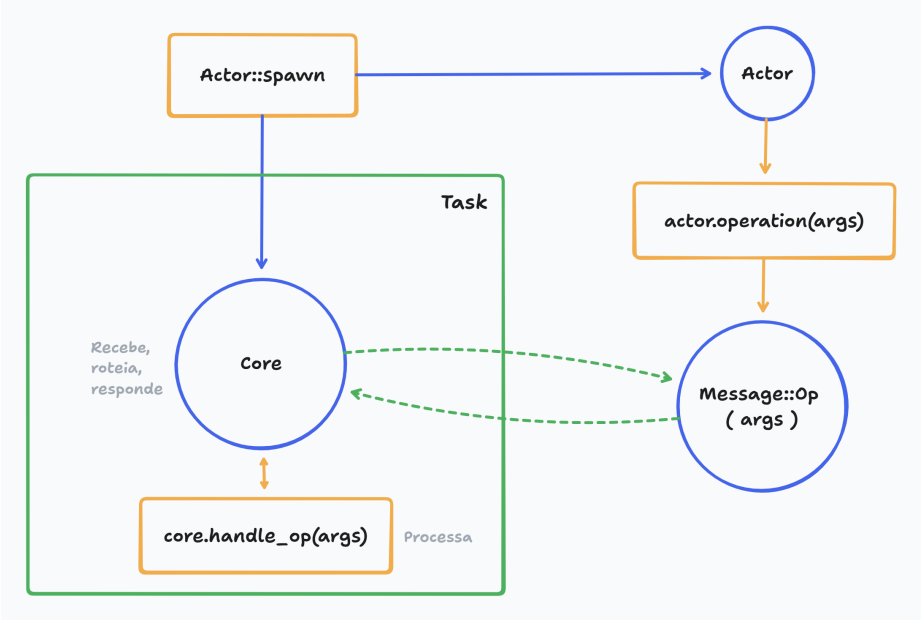

Figura 4.1: *Representação da estrutura de um ator*

Primeiramente, haverá um módulo para cada ator, e se um ator é subordinado à existência de outro ator pela lógica de negócio, ele pode ser criado como um submódulo.

Neste trabalho, o padrão para módulos será ter um arquivo e uma pasta com o mesmo nome (exceto pela extensão `.rs`). Por exemplo, para um ator chamado App, deve haver um `app.rs` e uma pasta `app/` no mesmo diretório.

Até aqui, três partes principais de um ator foram apresentadas:

- As mensagens que serão enviadas ao ator
- A lógica de processamento das mensagens recebidas
- O endereço que permite a comunicação entre atores

Essas três partes serão traduzidas para partes separadas do código Rust como:

- Um enum privado para definir os tipos das mensagens
- Uma struct privada para gerenciar o estado do ator e definir como lidar com cada mensagem
- Uma struct pública para abstrair o canal *mpsc* que serve como o endereço

Para o exemplo do ator App, haverá:

- Um enum `Message` em `app/message.rs` definindo o tipo para as mensagens
- Uma struct `Core` em `app/core.rs` que mantém a lógica do ator
- Uma struct `App` em `app.rs` para servir como a interface pública para dependentes deste ator

Onde `App` deve ser público globalmente, enquanto `Message` e `Core` ficam restritos ao super-módulo `app`.

Note que a convenção é que a struct que serve como interface pública para o ator será nomeada com o nome efetivo do ator para melhor legibilidade. Esta chamada interface pública deve fornecer métodos que abstraem a lógica de construção de mensagens e utilização de canais.

A seguir está destrinchado em mais detalhes o que estará em cada uma das 3 partes.

4.5.1 Mensagens

Assim como tudo em Rust, as mensagens enviadas através de canais devem ser tipadas. Caso um ator só execute um tipo de ação, ou seja, só receba um tipo de mensagem, então qualquer tipo pode ser usado para representar as mensagens. Caso ele lide com diferentes tipos de mensagens (o que acontece na maioria dos casos), um *enum* será usado.

No contexto do exemplo dado, o *enum* a ser criado no arquivo `app/message.rs` seria o demonstrado no programa 4.14.

Programa 4.14 Exemplo de mensagens tipadas.

```
1    enum Message {
2       SetThing { value: i32 },
3       GetThing { tx: oneshot::Sender<i32> }
4    }
```

A maior vantagem de tipar mensagens com *enums* é que o envio e o tratamento são verificados em tempo de compilação. Isso cria uma garantia forte sobre quais mensagens que podem estar envolvidas em uma dada comunicação e o compilador Rust garantirá que:

- O receptor fornecerá tratamento explícito para todas as mensagens possíveis
- O remetente nunca enviará uma mensagem inválida

As garantias acima não se aplicam à implementação do Modelo de Atores em Elixir.

4.5.2 Lógica

Toda a lógica de processamento de mensagens para um ator ficará dentro de uma *struct* chamada de `Core`. Ela será também onde será feito o gerenciamento de estado do ator e pelos detalhes específicos para criação da *task* e do canal `mpsc` dedicado para a comunicação.

No contexto do exemplo dado, a *struct* a ser criada no arquivo `app/core.rs` seria o exibido no programa 4.15.

Programa 4.15 Exemplo de struct Core para lógica do ator.

```
1   pub struct Core {
2      thing: i32,
3   }
```

O `Core` deve ser instanciado com a invocação do seu construtor, nesse momento as dependências do ator deverão ser injetadas como argumentos do construtor. O construtor será chamado `new` caso não possa falhar, caso contrário será chamado de `build`. Por exemplo, a inicialização de um ator pode depender do valor de uma variável de ambiente que talvez não exista, e isso leva a uma falha.

Programa 4.16 Exemplo de construtor.

```
1   impl Core {
2      pub fn new(thing: i32) -> Self {
3         Self {
4            thing
5         }
6      }
7   }
```

Deverá ser fornecido um método de instância `init` (veja o programa 4.17) responsável por conter a lógica de processamento das mensagens recebidas, delegando elas para os métodos responsáveis.

Note que o método `init` recebe `mut self` como primeiro argumento (sem uso de `&`), isso significa que ele consome a instância. Em outras palavras, aquele que tiver posse de uma instância do `Core` do App a perderá assim que `init` for invocada, impedindo que um mesmo objeto seja instanciado como 2 atores compartilhando o mesmo espaço de memória. O segundo parâmetro é a ponta de recebimento de mensagens do canal para se comunicar com o ator.

O bloco `while let Some(message) = rx.recv().await` é um *loop* que continua processando mensagens até que o canal seja fechado. Adicionalmente, cada ramo do bloco `match` apenas delega a execução para um método privado `handle_`. Em caso de necessidade de enviar resposta, isso é feito dentro do ramo, assim o método privado `handle_` é responsável apenas pela lógica de processamento da mensagem e não pela comunicação com o receptor.

4.5.3 Endereço

Para se aproveitar ao máximo dos recursos que a linguagem Rust oferece, é ideal a criação de um tipo que abstraia a lógica de uso dos canais, tanto para facilitar a leitura do código, quanto para diminuir duplicação de código. Ao contrário do que é feito em Elixir,

Programa 4.17 Exemplo de método init para processamento de mensagens.

```
impl Core {
   // ...
   pub async fn init(mut self, mut rx: mpsc::Receiver<Message>) {
      while let Some(message) = rx.recv().await {
         match message {
            Message::SetThing { value } => {
               self.handle_set_thing(value);
            }
            Message::GetThing { tx } => {
               let result = self.handle_get_thing();
               let _ = tx.send(result);
            }
         }
      }
   }

   fn handle_set_thing(&mut self, value: i32) {
      self.thing = value;
   }

   fn handle_get_thing(&self) -> i32 {
      self.thing
   }
}
```

onde se atribui o nome do ator ao módulo onde se encontra a lógica de processamento, aqui o nome será atribuído a essa abstração sobre o endereço.

No programa 4.18, a struct `App` encapsula o `Sender` do canal `mpsc` e fornece métodos públicos que abstraem completamente a lógica de envio de mensagens. O método `thing()` cria um canal `oneshot` para receber a resposta, envia a mensagem `GetThing` e aguarda a resposta. O método `set_thing()` simplesmente envia a mensagem `SetThing` com o valor fornecido.

Essa abstração permite que outros atores interajam com o ator sem precisar se preocupar com detalhes de comunicação por canais, mantendo a interface limpa e tipada.

4.6 Testabilidade

Uma das principais vantagens da arquitetura baseada em atores é a facilidade de testar componentes isoladamente. Como cada ator possui dependências injetadas através de suas interfaces, é possível substituir dependências reais por implementações mockadas durante os testes.

4.6.1 Mockall

Para simplificar a criação de mocks, este trabalho faz uso da biblioteca `mockall` (Somers, 2025), que gera automaticamente implementações *mock* a partir de traits ou structs. Isso

Programa 4.18 Exemplo de struct para endereço do ator.

```rust
use tokio::sync::{mpsc, oneshot};
use crate::error::{AppError, FatalActorError};

pub struct App {
    tx: mpsc::Sender<Message>,
}

impl App {
    pub fn spawn(value: i32) -> Self {
        let (tx, rx) = mpsc::channel(crate::BUFFER_SIZE);
        let core = Core::new(value);
        let _ = tokio::spawn(async move {
            core.init(rx).await;
        });

        Self { tx }
    }

    pub async fn thing(&self) -> Result<i32, AppError> {
        let (tx, rx) = oneshot::channel();
        let message = Message::GetThing { tx };

        self.tx.send(message).await.map_err(|_| AppError::Fatal(
            FatalActorError::ActorSendFailed {
                actor_name: "App",
                operation: "get thing".to_string(),
            }
        ))?;

        rx.await.map_err(|e| AppError::Fatal(
            FatalActorError::ActorRecvFailed {
                actor_name: "App",
                operation: "get thing".to_string(),
                source: e,
            }
        ))?
    }

    pub async fn set_thing(&self, value: i32) -> Result<(), AppError> {
        let message = Message::SetThing { value };

        self.tx.send(message).await.map_err(|_| AppError::Fatal(
            FatalActorError::ActorSendFailed {
                actor_name: "App",
                operation: "set thing".to_string(),
            }
        ))
    }
}
```

elimina a necessidade de escrever manualmente código *boilerplate* para *mocks* e garante que os *mocks* permaneçam sincronizados com as interfaces reais. O código para *mock* de um ator deve ser colocado no arquivo `mock.rs` do módulo do ator e o módulo deve ser declarado como no programa 4.19.

Programa 4.19 Exemplo de declaração de módulo para mock.

```
1    #[cfg(test)]
2    pub mod mock;
```

Desse modo, o código do mock não será incluso no binário de produção. Veja um exemplo no programa 4.20.

Programa 4.20 Exemplo de mock usando mockall.

```
1    mock!{
2       #[derive(Debug)]
3       pub App {
4          pub async fn operation(&self, params: Params) -> Result<Value>;
5       }
6
7       impl Clone for Actor {
8          fn clone(&self) -> Self;
9       }
10   }
```

A macro `mock!` do mockall gera automaticamente uma estrutura com nome `MockApp` que implementa todos os métodos especificados e permite configurar comportamentos esperados para cada invocação. Além disso, a macro gera implementações de traits como `Debug` e `Clone`, mantendo a compatibilidade com a interface original.

4.6.2 Integração de Mocks

Para permitir a substituição fácil entre implementações reais e mockadas, deve-se usar atributos de compilação condicional (`#[cfg(test)]` e `#[cfg(not(test))]`) para alternar entre as implementações. Veja um exemplo no programa 4.21.

Durante a compilação normal, a implementação real `OtherActor` é usada. Durante a compilação de testes, automaticamente substitui por `MockOtherActor`. Com o uso de compilação condicional, é possível implementar uma infraestrutura de testes bastante robusta que terá nenhum impacto no código de produção.

4.6.3 Testes Unitários

Com *mocks* em vigor, escrever testes unitários torna-se direto. Cada teste pode configurar o comportamento esperado de suas dependências *mockadas* e verificar que o ator sob teste se comporta corretamente. Veja um exemplo no programa 4.22.

Programa 4.21 Alternância entre implementações reais e mockadas.

```
#[cfg(not(test))]
use crate::other_actor::OtherActor;

#[cfg(test)]
use crate::other_actor::mock::MockOtherActor as OtherActor;

pub struct App {
    dependency: OtherActor,
}

impl App {
    pub fn new(dependency: OtherActor) -> Self {
        Self { dependency }
    }
}
```

Programa 4.22 Exemplo de teste unitário com mocks.

```
#[tokio::test]
async fn test_my_actor_operation() {
    let mut mock_dep = MockOtherActor::new();

    // Configura o comportamento esperado do mock
    mock_dep
        .expect_get_value()
        .times(1)
        .returning(|| Ok(42));

    let actor = App::spawn(mock_dep);

    let result = actor.operation().await;

    assert_eq!(result, Ok(42));
}
```

Os *mocks* permitem testar tanto cenários de sucesso quanto de falha, validando o tratamento de erros e garantindo que as interações entre atores funcionam corretamente em isolamento.

Capítulo 5

Estudo de Caso: patch-hub

Uma vez apresentadas as propostas de adaptação do Modelo de Atores para experimentação fora do domínio dos sistemas distribuídos, chega o momento de realizar um estudo de caso para aplicar a proposta em um projeto real para posterior análise de resultados.

O projeto escolhido foi uma ferramenta de software livre desenvolvida sob a organização Kworkflow (kw) chamada de patch-hub. Neste capítulo, será apresentado o contexto do projeto, a arquitetura original e suas limitações, os objetivos do estudo de caso e a proposta de refatoração do projeto com o uso do Modelo de Atores, cuja implementação estará disponível na íntegra no repositório do projeto no GitHub.

5.1 Contexto

Para compreender adequadamente o projeto patch-hub e sua proposta, é necessário entender o ecossistema de desenvolvimento do kernel Linux e os desafios enfrentados pelos desenvolvedores. O Linux é um sistema operacional de software livre baseado no kernel Unix-like desenvolvido por Linus Torvalds em 1991 (Wikipedia, 2025a). O kernel Linux é o núcleo do sistema operacional, responsável por gerenciar recursos de hardware, processos e comunicação entre componentes do sistema (Wikipedia, 2025b).

5.1.1 O Processo de Contribuição para o Kernel Linux

O desenvolvimento do kernel Linux é um processo colaborativo que envolve milhares de desenvolvedores ao redor do mundo. Devido à sua complexidade e tamanho, o kernel é organizado em subsistemas especializados, cada um com responsabilidades específicas (Linux Kernel Documentation, 2025b). Contribuições para o kernel, conhecidas como *patches*, são tradicionalmente enviadas através de listas de discussão públicas por email e revisadas por mantenedores especializados (Linux Kernel Documentation, 2025a).

Essa forma de contribuição é um processo tradicional e bem estabelecido, mas que traz diversos desafios para os desenvolvedores. Os desenvolvedores precisam se inscrever em múltiplas listas de email e gerenciar um volume considerável de mensagens. Isso dificulta muito acompanhar o status de suas contribuições.

5.1.2 O Projeto lore.kernel.org

Para mitigar essas dificuldades, foi criado o projeto lore.kernel.org, um arquivo público que fornece uma interface web para visualizar e pesquisar mensagens das listas de discussão do kernel (Tadokoro, 2025). O lore.kernel.org oferece uma *API* que permite acesso programático aos *patches* arquivados, incluindo funcionalidades de busca, paginação e filtros avançados.

A *API* do lore suporta consultas complexas através de parâmetros de *query string*, permitindo filtrar *patches* por critérios como assunto, autor, data e conteúdo. Esta funcionalidade é fundamental para ferramentas que buscam automatizar ou simplificar a interação com o ecossistema de *patches* do kernel.

5.1.3 O Projeto Kworkflow

O projeto Kworkflow (kw) surgiu como uma resposta a complexidade e desafios de configuração do ambiente de desenvolvimento para o kernel Linux (KWorkflow, 2025a). O kw é descrito como uma "ferramenta de fluxo de trabalho de desenvolvedor de kernel" que visa reduzir a sobrecarga de configuração e fornecer ferramentas para tarefas comuns do desenvolvimento de kernel.

O projeto oferece um utilitário de terminal unificado para diversas operações como compilação, deploy, debug e gerenciamento de configurações, consolidando ferramentas anteriormente dispersas.

5.1.4 O patch-hub

O patch-hub é uma ferramenta desenvolvida como parte do ecossistema Kworkflow, especificamente projetada para interagir com a *API* do lore.kernel.org (KWorkflow, 2025b). Sua missão é agilizar a navegação e revisão de contribuições para o kernel Linux, fornecendo uma interface de usuário baseada em terminal (*TUI*) que simplifica o processo de descoberta, análise e acompanhamento de *patches*.

A ferramenta permite aos desenvolvedores navegar por listas de patches, aplicar filtros de busca, visualizar conteúdo de patches e gerenciar seu fluxo de trabalho de revisão diretamente do terminal, eliminando a necessidade de alternar entre interfaces web e ferramentas de linha de comando.

5.2 Arquitetura Original

A implementação original do patch-hub seguia uma arquitetura majoritariamente *single-threaded* e monolítica com inspirações no padrão arquitetural MVC. A organização do código era dada em termos dos seguintes módulos:

- **`lore`**: Interação com a *API* do lore.kernel.org
- **`ui`**: Renderização de terminal e lógica de exibição
- **`cli`**: *Parsing* de argumentos de linha de comando

- **app**: Estado central da aplicação e gerenciamento de dados
  - `logger`: Gerencia funcionalidade de *logging*
  - `popup`: Cria e gerencia diálogos *popup*
  - `config`: Gerencia configurações da aplicação
- **handler**: Manipulação de entrada do usuário e processamento de eventos

5.2.1 Funcionalidades Principais

A implementação do patch-hub (tanto a original quanto a refatorada) fornece um conjunto de funcionalidades para interagir com o arquivo de *patches* do kernel Linux. O sistema permite a seleção e navegação de listas de discussão arquivadas no *lore.kernel.org*, apresentando os conjuntos de *patches* associados. Ao examinar uma série específica, a interface disponibiliza o conteúdo individual dos *patches* juntamente com metadados relevantes, incluindo título, autor, versão, quantidade de *patches* e *timestamp* da última atualização.

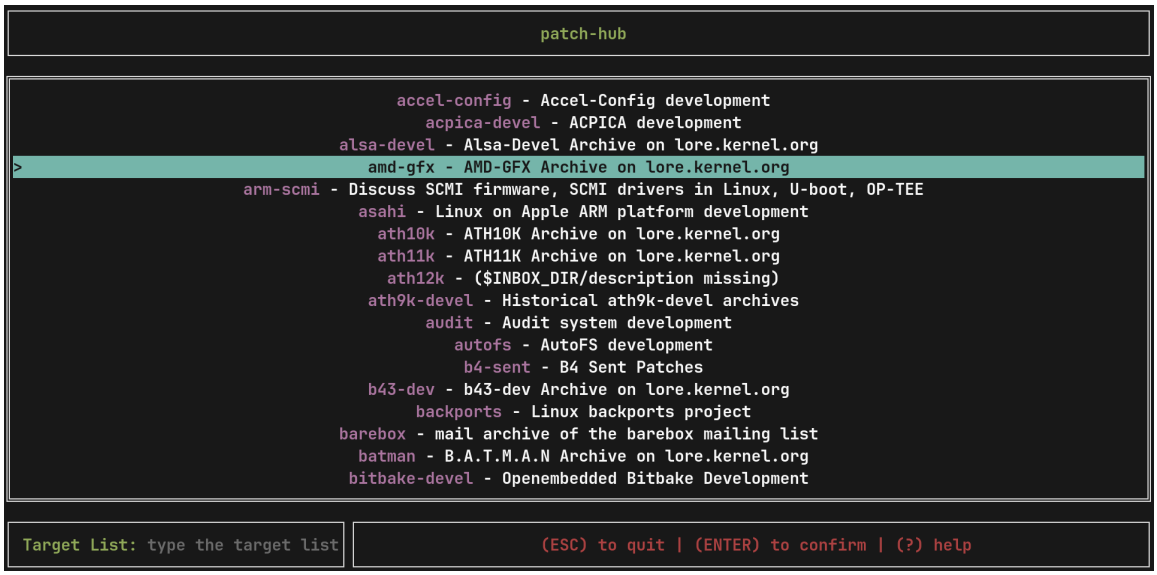

Figura 5.1: *Interface do patch-hub.*

O sistema implementa operações fundamentais para o *workflow* de revisão de *patches*: aplicação de *patches* em repositórios locais do kernel, organização de séries em um sistema de favoritos, e composição de respostas com tags de revisão padrão, como `Reviewed-by`.

A renderização do conteúdo dos patches é implementada com suporte a múltiplos backends, incluindo ferramentas externas, além de um renderizador padrão. Esta abordagem permite variação na apresentação visual dos patches ao gosto do usuário sem comprometer a portabilidade da aplicação.

5.3 Limitações da Arquitetura Original

Apesar de sua funcionalidade, a implementação original do patch-hub apresentava limitações arquiteturais fundamentais que comprometiam sua extensibilidade, manutenibilidade e capacidade de evolução. Essas limitações vinham de decisões de design que, embora funcionais para o escopo inicial, criavam gargalos estruturais que dificultavam a adição de novas funcionalidades e a manutenção do código no longo prazo.

5.3.1 Gargalo de Mutação Centralizada

O principal problema arquitetural reside na concentração de todo o estado da aplicação em uma única estrutura `App` que deve ser passada como `&mut App` através de todo o código. Esta abordagem cria um *gargalo de mutação centralizada* que viola princípios de *borrow checking* do Rust.

O sistema de *ownership* do Rust impõe que quando uma função recebe `&mut App`, ela adquire uma referência mutável exclusiva de toda a estrutura. Isso significa que:

- Nenhum outro código pode ler ou escrever qualquer campo de `App` até que essa referência seja liberada
- Campos não podem ser emprestados independentemente enquanto `&mut App` está ativa
- Componentes internos não podem modificar componentes irmãos sem liberar a referência primeiro

Esta restrição força a centralização de toda lógica de mutação nos métodos de `App`, impedindo que componentes gerenciem seu próprio estado de forma independente.

O resultado é uma violação do princípio da responsabilidade única, forçando a estrutura central a conhecer detalhes de implementação de todos os seus componentes. Além disso, o princípio da inversão de dependência também é violado, fazendo com que a estrutura de alto nível dependa diretamente de mecanismos de baixo nível de seus componentes internos.

5.3.2 Impossibilidade de Composição Modular

O design atual impede a composição modular de *handlers* e componentes devido às restrições de *borrowing*. *Handlers* de eventos devem receber `&mut App` inteiro mesmo quando precisam acessar apenas um subconjunto específico de dados. Isso cria uma situação onde *handlers* não podem ser decompostos em funções auxiliares menores e também funções auxiliares que precisam de múltiplos campos de `App` enfrentam conflitos por tentarem pegar referências a partes de algo que já possui uma referência mutável.

Esta limitação força a criação de métodos monolíticos em `App` que conhecem detalhes de implementação de múltiplos componentes, aumentando a complexidade e reduzindo a testabilidade.

5.3.3 Atualizações de Estado Intercomponentes Problemáticas

A arquitetura torna impossível a atualização direta de estado entre componentes, forçando todas as interações através da estrutura central `App`. Quando um componente precisa modificar o estado de outro componente irmão, ele deve passar pela estrutura `App` como intermediário, criando acoplamento forte entre componentes que deveriam ser independentes.

5.3.4 Concorrência e Responsividade Limitadas

A natureza *single-threaded* da aplicação cria gargalos significativos durante operações de I/O, resultando em:

- Interfaces não responsivas durante requisições de rede
- Bloqueio da UI durante operações de sistema de arquivos
- Lógica complexa para implementar telas de carregamento
- Impossibilidade de realizar operações paralelas independentes

Para contornar esse problema, foi feito uso de mecanismos para paralelizar certas regiões do código. A complicação aqui é que referências em Rust não podem ser compartilhadas entre *threads* pois não há garantias de que uma *thread* vá ser encerrada antes do dono do valor referido, o que poderia causar vazamentos de memória ou condições de corrida indesejáveis.

5.3.5 Dificuldades de Testabilidade

A arquitetura monolítica torna extremamente difícil a criação de testes unitários eficazes. Testes de *handlers* individuais requerem a construção de toda a estrutura `App`, incluindo dependências que podem não ser relevantes para o teste específico. Isso resulta em:

- Testes lentos devido à necessidade de inicialização completa
- Testes frágeis que quebram quando componentes não relacionados são modificados
- Impossibilidade de isolar comportamento específico para teste
- Dificuldade em criar *mocks* e *stubs* para dependências externas

Esta limitação compromete significativamente a capacidade de garantir qualidade através de testes automatizados, aumentando o risco de regressões em mudanças futuras.

5.4 Objetivos da Reimplementação

Fica evidente que o patch-hub necessita de refatorações para tentar resolver os problemas supracitados. Em especial, fica notável uma grande necessidade de melhorias do ponto de vista da flexibilidade do sistema. Tendo em vista que o Modelo de Atores proporciona alta modularidade e baixo acoplamento aos sistemas, isso torna o patch-hub um candidato

interessante para experimentação e avaliação de valor da proposta. Além disso, o patch-hub toca em domínios suficientemente diversos para que o experimento seja bastante completo.

Este trabalho não criará uma reescrita completa com 100% de compatibilidade de funcionalidades. Ao invés disso, um subconjunto das funcionalidades foi escolhido para implementação, suficiente para ter as funcionalidades principais do patch-hub, que são:

- Uma interface de usuário de texto interativa
- Integração com *API* do Lore com suporte a *cache*
- Navegar através das listas de discussão catalogadas
- Acesso a um feed de patches enviados para uma lista de discussão
- Fornecer uma visualização do conteúdo e metadados de um patch
- Configurações personalizáveis
- Um sistema de log

5.5 Modelagem com Atores

Com objetivos estabelecidos, foi feita uma modelagem da arquitetura do patch-hub com o uso de atores. A modelagem adota os princípios SOLID como guias. Os atores serão categorizados em 2 grupos considerando o tipo de interação que eles são responsáveis: externas ou internas.

O primeiro grupo é formado pelos atores responsáveis por interações com componentes externos ao sistema, como, por exemplo, o sistema de arquivos, a internet ou o terminal. Este grupo será chamado de atores de integração pois permitem a integração do programa com outros sistemas, gerando, assim, efeitos colaterais no mundo real.

O segundo grupo é formado pelos atores que irão interagir apenas com demais atores, ou seja, componentes internos ao sistema. Este grupo tem uma diferença fundamental no que diz respeito a sua testabilidade.

Por dependerem apenas de outros atores, testes unitários se tornam triviais uma vez que se criem *mocks*. Ao testar um ator contra *mocks* que imitem o comportamento desejado dos atores de dependência, dá-se a garantia de que apenas o código do próprio ator é o que influencia no resultado dos testes. Este grupo será chamado de atores internos por não interagirem diretamente com sistemas externos.

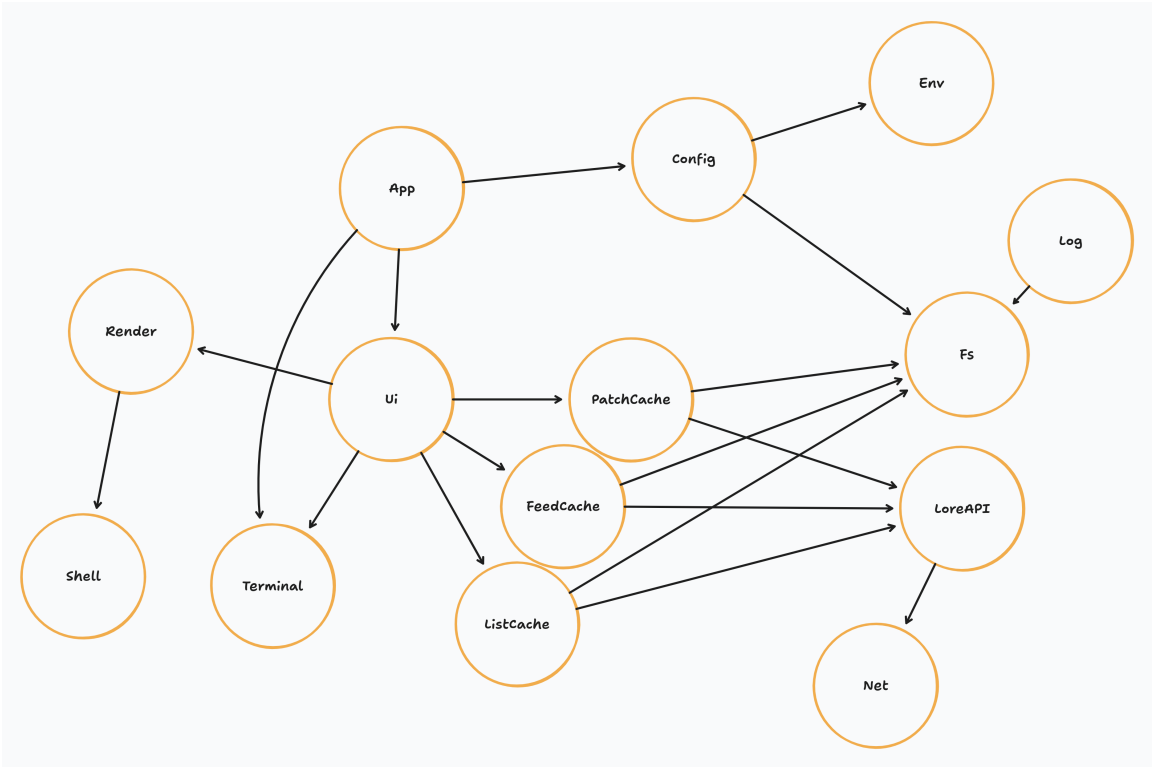

Figura 5.2: *Diagrama de atores do patch-hub*

5.5.1 Atores de Integração

Primeiramente, foi catalogado quais capacidades do patch-hub dependem unicamente de interação com sistemas externos ao programa que serão inclusas na versão reescrita do projeto. As capacidades escolhidas foram:

- Operações do sistema de arquivos
- Requisições HTTP
- Manipulação de variáveis de ambiente
- Controle de entrada e saída do terminal
- Execução de comandos externos

Essas capacidades não fazem parte da lógica de negócio do patch-hub, mas servem de bases para permitir a sua implementação. Cada uma delas pode ser traduzida para um ator em nossa aplicação seguindo os princípios SOLID, especialmente os princípios de responsabilidade única e de segregação de interfaces.

Sistema de Arquivos

O ator Sistema de Arquivos é responsável por gerenciar as operações de leitura, escrita, remoção e listagem de arquivos e diretórios. Ele é responsável por garantir que as operações sejam realizadas de forma segura, garantindo que o sistema de arquivos seja consistente e que os arquivos e diretórios sejam gerenciados corretamente.

Implementação: `src/fs.rs`

Mensagens:

- `ReadFile`: Abre um arquivo apenas para leitura (não cria se não existir)
- `WriteFile`: Abre um arquivo para escrita (trunca conteúdo, cria se necessário)
- `AppendFile`: Abre um arquivo para anexar (cria se necessário)
- `RemoveFile`: Remove um arquivo do sistema de arquivos
- `ReadDir`: Lê o conteúdo de um diretório
- `MkDir`: Cria um diretório e seus pais
- `RmDir`: Remove um diretório e seu conteúdo

Rede

O ator Rede é responsável por gerenciar as operações de requisição HTTP. Ele é o mecanismo base que vai permitir a integração do sistema com a *API* web do lore.kernel.org.

Implementação: `src/net.rs`

Depende:

- **Configurações**: Para obter configurações de timeout e outras configurações de rede
- **Log**: Para registrar operações de rede para posterior análise

Mensagens:

- `Get`: Executa uma requisição HTTP GET para a URL especificada com headers opcionais
- `Post`: Executa uma requisição HTTP POST para a URL especificada com headers opcionais e corpo
- `Put`: Executa uma requisição HTTP PUT para a URL especificada com headers opcionais e corpo
- `Delete`: Executa uma requisição HTTP DELETE para a URL especificada com headers opcionais e corpo
- `Patch`: Executa uma requisição HTTP PATCH para a URL especificada com headers opcionais e corpo

Variáveis de Ambiente

O ator Variáveis de Ambiente é responsável por gerenciar as operações de definição, remoção e recuperação de variáveis de ambiente.

Implementação: `src/env.rs`

Mensagens:

- `Set`: Define uma variável de ambiente para um valor especificado
- `Unset`: Remove uma variável de ambiente
- `Get`: Recupera o valor de uma variável de ambiente (retorna VarError se não encontrada)

Terminal

O ator Terminal gerencia o *loop* de eventos da interface de usuário baseada na biblioteca Cursive. Ele mantém uma fila FIFO interna de eventos de *UI* (teclas pressionadas, seleções, etc.) além de ser responsável por desenhar no terminal. Ele também expõe uma *API* baseada em mensagens para atualizar a interface e recuperar os eventos de *UI* ocorridos.

Implementação: `src/terminal.rs`

Depende:

- **Log**: Para registrar eventos do terminal para posterior análise

Mensagens:

- `Show`: Atualiza a tela exibida com um novo tipo de tela
- `GetUiEvent`: Recupera o próximo evento de *UI* da fila
- `ClearUiEvents`: Limpa a fila de eventos de *UI*
- `Quit`: Termina a *UI* e sai da aplicação

O método `spawn()` do ator Terminal retorna uma tupla contendo a interface do ator e um `JoinHandle` que permite que outros atores detectem quando o terminal foi encerrado, permitindo encerramento gracioso da aplicação. Este design evita a necessidade de envolver o `JoinHandle` em um `Arc` e simplifica o gerenciamento do ciclo de vida do terminal.

Shell

O ator Shell é responsável por executar comandos integrando o patch-hub ao shell do sistema hospedeiro. Através dele é possível expandir as funcionalidades do patch-hub com o uso de ferramentas externas.

Implementação: `src/shell.rs`

Depende:

- **Log**: Para registrar operações de execução de comandos para posterior análise

Mensagens:

- `Execute`: Executa um programa externo com comando, argumentos e entrada stdin opcionais retornando a saída padrão do programa

5.5.2 Atores Internos

As demais capacidades foram catalogadas. Dessa vez são capacidades de alto nível, ou seja, que não dependem de detalhes de implementação específicos. Elas vão depender das abstrações feitas para os atores de integração, endereçando, assim, o DIP. As capacidades escolhidas foram:

- Registrar eventos com um sistema de logs
- Gerenciar configurações
- Acessar a Lore API
- Suporte a caching das requests feitas
- Desenhar a interface do usuário (TUI)

Logs

Fornece um serviço de *logging* centralizado e *thread-safe*. Ele recebe mensagens de log de outros atores e as escreve em um arquivo de log designado, fornecendo diferentes níveis de log para *debugging* e monitoramento.

Implementação: `src/log.rs`

Depende:

- **Sistema de Arquivos**: Para persistir logs em um arquivo

Mensagens:

- `Log`: Registra uma mensagem com um nível de log específico (Info, Warning, Error) através de uma estrutura `LogMessage`
- `Flush`: Escreve as mensagens de log em buffer para stderr e encerra o logger
- `CollectGarbage`: Executa a limpeza de arquivos de log antigos baseado na idade máxima configurada

Configurações

Responsável por carregar configurações de um arquivo de configuração, fornecê-las a outros atores sob demanda e persistir mudanças.

Implementação: `src/app/config.rs`

Depende:

- **Sistema de Arquivos**: Para salvar e carregar o arquivo de configuração
- **Variáveis de Ambiente**: Para obter os valores padrão das configurações

Mensagens:

- `Load`: Recarrega a configuração do arquivo fonte
- `Save`: Salva a configuração atual no arquivo fonte

- `GetPath`: Recupera o valor de uma configuração baseada em caminho (ex: LogDir, CachePath)
- `SetPath`: Atualiza o valor de uma configuração baseada em caminho
- `GetLogLevel`: Recupera o nível de log atual
- `SetLogLevel`: Atualiza o nível de log
- `GetUSize`: Recupera o valor de uma configuração numérica (ex: MaxAge, Timeout)
- `SetUSize`: Atualiza o valor de uma configuração numérica
- `GetRenderer`: Recupera a configuração de um renderizador (ex: PatchRenderer)
- `SetRenderer`: Atualiza a configuração de um renderizador

Lore API

Gerencia toda a comunicação com a *API* externa do Lore patchwork. Ele abstrai os detalhes das requisições HTTP e *parsing* de dados, fornecendo uma interface limpa e tipada para buscar dados como listas de discussão, séries de *patches* e *patches* individuais. Ele atua como um tradutor entre as estruturas de dados internas da aplicação e as respostas brutas da *API*. Ele não realiza nenhum *cache* por si só.

Implementação: `src/api/lore.rs`

Depende:

- **Rede**: Para executar as requisições HTTP subjacentes

Mensagens:

- `GetAvailableLists`: Busca todas as listas de discussão disponíveis (sem paginação)
- `GetAvailableListsPage`: Busca uma lista paginada de listas de discussão disponíveis
- `GetPatchFeedPage`: Busca um feed paginado de patches para uma lista de discussão específica
- `GetRawPatch`: Busca o conteúdo bruto de um único patch em formato texto
- `GetPatchHtml`: Busca o conteúdo HTML de um único *patch*
- `GetPatchMetadata`: Busca os metadados de um único *patch* em formato JSON

Interface de Usuário

O ator UI gerencia a lógica de navegação e estado da interface, coordenando a comunicação entre os caches e o Terminal. Ele mantém o estado de navegação (lista atual, página, seleção) e decide quais telas mostrar com base nas ações do usuário.

Implementação: `src/app/ui.rs`

Depende:

- **Terminal**: Para atualizar a interface visual
- **Cache de Listas**: Para recuperar dados de listas de discussão
- **Cache de Feed**: Para recuperar metadados de patches
- **Cache de Patches**: Para recuperar conteúdo de patches
- **Renderizador**: Para renderizar conteúdo de patches
- **Log**: Para registrar operações de UI

Mensagens:

- `ShowLists`: Exibe a visão de listas de discussão na página especificada
- `ShowFeed`: Exibe o feed de patches para uma lista específica na página especificada
- `ShowPatch`: Exibe o conteúdo renderizado de um patch específico com título, lista e message ID
- `UpdateSelection`: Atualiza o item selecionado na visão atual (sem retorno)
- `PreviousPage`: Navega para a página anterior na visão atual
- `NextPage`: Navega para a próxima página na visão atual
- `NavigateBack`: Navega de volta na hierarquia (patch, depois feed, depois listas; nas listas, encerra a aplicação)
- `SubmitSelection`: Submete a seleção atual, retornando uma ação de navegação (abre feed, abre patch, ou encerra)
- `GetState`: Recupera o estado atual da UI

Renderizador

O ator Render é responsável por processar conteúdo bruto de patches e convertê-los em uma representação visual formatada. Ele suporta múltiplos backends de renderização, incluindo ferramentas externas como `bat` ou `delta`.

Implementação: `src/render.rs`

Depende:

- **Shell**: Para executar programas externos de renderização
- **Configurações**: Para obter a configuração do renderizador a ser usado

Mensagens:

- `Render`: Renderiza o conteúdo bruto do patch usando o renderizador configurado

Cache

Implementação: `src/app/cache.rs`

A fim de agilizar o acesso a dados diminuindo a quantidade de operações que dependam de chamadas de *API* ou do sistema de arquivos, será feito o uso de técnicas de *caching*.

Para cada domínio de dados, será criado um ator específico que se encarregará de gerenciar o *cache* para esse domínio. Sendo responsável por lidar com *cache misses* e *cache hits*, assim como com a validação de *cache*, garantindo consistência dos dados realizando o mínimo de acessos necessários a sistemas externos.

Listas de Discussão

Armazena em *cache* a lista de todas as listas de discussão disponíveis da *API* do Lore. Ele busca a lista completa, ordena alfabeticamente e a persiste no sistema de arquivos para acelerar inicializações subsequentes da aplicação. Ele gerencia validação de *cache* para atualizar periodicamente a lista.

Implementação: `src/app/cache/mailing_list.rs`

Depende:

- **Lore API**: Para buscar os dados da lista de discussão
- **Sistema de Arquivos**: Para persistir o cache em disco
- **Configurações**: Para obter o caminho do arquivo de cache

Mensagens:

- `Get`: Recupera uma única lista de discussão por seu índice na lista ordenada
- `GetSlice`: Recupera um intervalo de listas de discussão para paginação
- `Refresh`: Força uma atualização completa do cache da API
- `Invalidate`: Limpa o cache em memória e em disco
- `IsAvailable`: Verifica se um intervalo de índices está disponível em cache
- `Len`: Retorna o número total de listas de discussão em cache
- `Persist`: Persiste o cache no sistema de arquivos
- `Load`: Carrega o cache do sistema de arquivos

Feed

Fornece *cache* por lista de discussão de metadados de *patches* (como autor, assunto e data). Ele busca *feeds* de *patches* sob demanda, suporta paginação e persiste os metadados no sistema de arquivos. Ele usa uma estratégia de validação inteligente para buscar apenas *patches* novos quando um *feed* foi atualizado no servidor.

Implementação: `src/app/cache/feed.rs`

Depende:

- **Lore API**: Para buscar metadados de patches
- **Sistema de Arquivos**: Para persistir o cache em disco

- **Configurações**: Para obter o caminho do diretório de cache
- **Log**: Para registrar operações de cache

Mensagens:

- `Get`: Recupera metadados de um único *patch* por seu índice dentro de um *feed* de lista de discussão
- `GetSlice`: Recupera um intervalo de metadados de *patches* para paginação
- `Refresh`: Atualiza inteligentemente o *cache* para uma lista de discussão específica, buscando apenas itens novos
- `Invalidate`: Limpa o *cache* para uma lista de discussão específica
- `IsAvailable`: Verifica se um intervalo de índices está disponível em *cache* para uma lista
- `Len`: Retorna o número de entradas de metadados em *cache* para uma lista
- `Persist`: Persiste o *cache* de uma lista no sistema de arquivos
- `Load`: Carrega o *cache* de uma lista do sistema de arquivos
- `IsLoaded`: Verifica se o *cache* de uma lista foi carregado do disco

Patches

Armazena em *cache* o conteúdo bruto de *patches* individuais (formato `.mbox`). Como o conteúdo do *patch* é imutável, uma vez que um *patch* é buscado e armazenado em *cache*, ele é considerado válido para sempre. Este ator armazena cada *patch* em um arquivo separado no disco e usa um *buffer* LRU (*Least Recently Used*) em memória para fornecer acesso rápido a *patches* visualizados recentemente.

Implementação: `src/app/cache/patch.rs`

Depende:

- **Lore API**: Para buscar o conteúdo bruto do patch
- **Sistema de Arquivos**: Para armazenar arquivos de patch permanentemente
- **Configurações**: Para obter o caminho do diretório de cache
- **Log**: Para registrar operações de cache

Mensagens:

- `Get`: Recupera o conteúdo bruto de um *patch* dado sua lista de discussão e ID de mensagem (busca da *API* se não estiver em *cache*)
- `Invalidate`: Remove um *patch* específico do *cache* em disco e *buffer* em memória
- `IsAvailable`: Verifica se um *patch* está em *cache* sem buscá-lo

App

O ator App atua como entidade central da aplicação, gerenciando o ciclo de vida e estado da aplicação. Ele é responsável pela inicialização da aplicação, processamento de eventos de UI do Terminal e encerramento gracioso quando apropriado.

Implementação: `src/app.rs`

Depende:

- **Mailing List Cache**: Para gerenciar o cache de listas de discussão
- **Feed Cache**: Para gerenciar o cache dos feeds das listas de patches
- **Terminal**: Para receber eventos de UI capturados pelo terminal
- **UI**: Para atualizar a interface de usuário com base nas ações do usuário
- **Log**: Para registrar operações da aplicação

Mensagens:

- `Shutdown`: Inicia o processo de encerramento graceful da aplicação

Note que os eventos de teclado são registrados pelo Terminal e processados pelo App. Todavia isto não é feito com uma mensagem enviada pelo Terminal para o App, mas sim o App que vai solicitar os eventos de *UI* ao Terminal no momento apropriado. Caso contrário, existiria um ciclo de dependências entre o Terminal e o App. Pois o App depende de Ui, que por sua vez depende de Terminal. Caso o Terminal tivesse de enviar mensagens ao App, então o Terminal dependeria do App, fechando o ciclo. Na presente proposta, as dependências precisam ser organizadas sem ciclos.

Capítulo 6

Análise de Resultados

Por fim, foi feita uma análise qualitativa e quantitativa da aplicação produzida. O foco é em analisar o impacto da arquitetura proposta no sistema, buscando entender quais benefícios foram obtidos e quais custos foram incorridos.

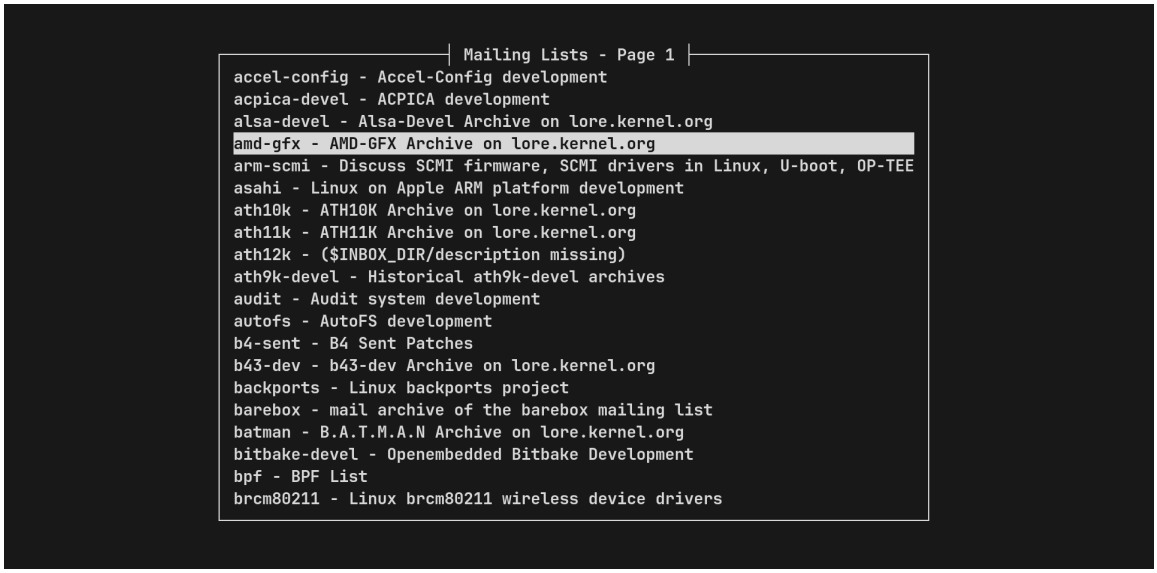

Figura 6.1: *Interface do Patch-Hub reescrito com o Modelo de Atores.*

6.1 Desempenho

De uma análise de desempenho, o grande ganho potencial da abordagem com o Modelo de Atores é o fato do paralelismo nativo. Considerando que cada ator é executado em uma *task* diferente, é possível executar código concorrentemente. Em especial, a biblioteca Tokio permite a alocação de *tasks* em mais do que uma *thread* do sistema operacional, o que permite a execução de código paralelo.

Por outro lado, o mecanismo de passagem de mensagens entre atores possui uma sobrecarga em tempo de execução. Como cada mensagem é encapsulada e enviada através

de um canal, isso representa uma indireção que requer diversas operações de sistema para ser realizada. Evidentemente, isso é mais lento que uma chamada de método direta.

Em resumo, a abordagem do Modelo de Atores dá um grande potencial para melhora de *performance*. Porém, se não for corretamente explorado, o custo de comunicação entre atores pode ser um problema e piorar a *performance* do sistema.

6.2 Extensibilidade

Extender funcionalidades do sistema é uma tarefa bastante simples. Como o código todo segue padrões de design bem definidos, é possível adicionar novas funcionalidades sem a necessidade de grandes refatorações.

Um exemplo dessa facilidade é a adição de novos atores. Para adicionar um novo ator, basta criar um novo módulo seguindo o padrão estabelecido, instanciá-lo e injetá-lo como dependência dos atores que vão usufruir de suas funcionalidades.

Também criar uma nova mensagem é muito simples. Apenas é necessário adicionar uma nova variante ao *enum* de mensagens do ator correspondente, atualizar o bloco `match` que lida com as mensagens recebidas pelo ator para chamar o novo *handler* e implementar um método na interface pública do ator para que outros atores possam enviar essa nova mensagem.

6.3 Testabilidade

A abordagem do Modelo de Atores facilita imensamente a criação de testes unitários. Como cada ator, seguindo o SRP, possui um domínio bem definido, é fácil criar *mocks* para suas dependências. Desse modo, é possível testar cada ator isoladamente, garantindo que o sistema comporte-se corretamente.

Um dos principais ganhos desse isolamento é a facilidade de se testar atores diferentes em paralelo, realizando testes com resultados determinísticos e rápidos.

Um exemplo dessa melhoria pode ser notado comparando a abordagem anterior para testes do antigo Patch-Hub com foco nas variáveis de ambiente. Como a versão antiga não isolava o gerenciamento de variáveis de ambiente, os testes quando executados em paralelo podiam ser afetados por condições de corrida ao modificarem o ambiente de execução. A solução para isso no projeto original foi forçar a execução em série de alguns testes que eram dependentes de variáveis de ambiente.

Na versão reescrita, os testes que vão interagir (direta ou indiretamente) com variáveis de ambiente recebem uma instância *mock* do ator Variáveis de Ambiente que é completamente independente das demais instâncias. Com isso, todos os testes podem ser executados em paralelo sem risco de condições de corrida ou qualquer outro tipo de interferência.

Quantitativamente, a facilidade de testabilidade pode ser medida pela cobertura de testes que saiu de 16.53% para 48.95%. Evidente que a reescrita não corresponde exatamente à implementação original, todavia, o aumento de cobertura é significativo.

6.4 Experiência do Desenvolvedor

O mais notável ganho no que toca a experiência do desenvolvedor é a facilidade de compreensão e modificação do código. Tendo-se isolado as responsabilidades em unidades menores e mais focadas, fica mais fácil entender o propósito de cada parte do código.

Por outro lado, existem alguns custos a serem pagos se tratando da transparência do código em si. Observe a seguinte invocação de método realizada em um *handler* do ator Lore API:

Programa 6.1 Exemplo de envio de mensagem para um ator.

```
1       let response = self
2           .net
3           .get(ArcStr::from(&url), Some(headers))
4           .await?;
```

Existe uma dificuldade de encontrar o código que será executado ao se enviar essa mensagem. Note que o método `get` do ator Rede é apenas um *wrapper* para fazer o uso de um canal para enviar a mensagem e receber a resposta, logo, sua definição não informa ao usuário o que de fato será executado.

Para poder encontrar o código que será executado, é necessário navegar manualmente pelo código fonte encontrando o módulo `net/core.rs` e encontrar o método `handle_get`. Essa indireção torna o código opaco para o usuário, além de evidentemente criar uma sobrecarga em tempo de execução.

6.5 Complexidade do Código

Uma diferença a ser considerada é o tamanho da base de código. A versão reescrita possui aproximadamente o dobro do tamanho da versão original, apesar de não ser uma implementação completa. Isso é um forte indicativo de que a abordagem do Modelo de Atores tende a produzir uma base de código maior. Apesar disso não necessariamente ser um problema, uma base de código maior carece de cuidados especiais para manter a qualidade e a legibilidade a longo prazo.

Outra dificuldade crucial a ser considerada está no uso de código assíncrono. Rust é uma linguagem que foi projetada para ser segura em primeiro lugar, o que implica numa série de asserções de segurança que devem ser feitas para garantir que o código assíncrono seja usado corretamente. O suporte a programação assíncrona em Rust ainda está em desenvolvimento, onde várias ferramentas da linguagem ainda não estão perfeitamente maduras. Isso não foi um impeditivo para o desenvolvimento deste trabalho, mas poderia ser em um projeto diferente e mais complexo.

Capítulo 7

Conclusão

O principal resultado deste trabalho demonstra que a adequação de padrões arquiteturais pode ser surpreendentemente flexível. Mesmo utilizando um padrão aparentemente inapropriado para o contexto, foi possível extrair valor significativo da abordagem ao reimaginar tanto o problema quanto o padrão, permitindo que ambos se encaixassem de forma interessante. Naturalmente, esse processo de adaptação demandou planejamento cuidadoso e múltiplas iterações.

Conclui-se que, embora os padrões arquiteturais possuam domínios naturais de aplicação, eles não estão rigidamente limitados a esses contextos. A exploração criativa de padrões em domínios distantes de sua origem pode resultar em soluções bem-sucedidas, desde que haja disposição para adaptar e reimaginar as abordagens estabelecidas.

7.1 Trabalhos Futuros

Por fim, ficam sugestões para trabalhos futuros considerando pontos da proposta que não foram implementados e que podem ser melhorados.

7.1.1 Completar a implementação

Como já foi mencionado, a reescrita do Patch-Hub não é uma implementação completa. Portanto, um trabalho futuro óbvio a ser desenvolvido seria concluir a implementação do sistema.

As principais funcionalidades que faltam são:

- **Gerenciamento de favoritos:** Não foi implementado uma lista de favoritos para navegação rápida.
- **Agrupamento de séries de *patches*:** A fim de simplificar a implementação, os *patches* são exibidos de forma individual. Na versão original, os *patches* são agrupados em séries (similar a como *commits* são agrupados em uma *branch*)

- **Operações com *patches*:** A tela de conteúdo de um *patch* no Patch-Hub original conta com *features* como: marcar como revisado, aplicar série de *patches* em um repositório local, entre outras.
- **Configurações:** A tela de edição de configurações não foi implementada.
- **Popups:** O sistema de *popups* como um todo não foi trabalhado.

7.1.2 Explorar mais combinações

Um segundo trabalho futuro óbvio seria seguir o mesmo espírito explorador deste trabalho. A ideia seria procurar um padrão arquitetural de propósito específico e aplicar a um domínio diferente.

Evidentemente que a chance do trabalho ser bem sucedido depende de uma análise prévia tentando procurar algum tipo de benefício do padrão escolhido que poderia ser aproveitado de forma criativa no novo contexto.

7.1.3 Gerenciamento de Ciclos de Dependências

O primeiro ponto a ser considerado é o gerenciamento de ciclos de dependências. Neste trabalho, não é oferecida uma solução universal para ciclos de dependências. No escopo do trabalho, houve duas situações que geraram ciclos de dependências:

- Ciclos de dependência entre o ator Log e o ator Fs.
- Ciclos de dependência entre o ator Terminal e o ator App.

Para o primeiro caso, o Log depende do Fs para persistir as mensagens em arquivos de log. Por sua vez, o Fs depende do Log para registrar erros de operações de arquivos.

Infelizmente, a "solução" adotada foi a de abrir mão do registro de erros de operações de arquivos no ator Fs diretamente. Ao invés disso, ele apenas retorna valores do tipo `Result<T, FsError>` para quem solicitar algum serviço e esse, por sua vez, seria responsável por registrar o erro caso ocorra.

No segundo caso, existe um ciclo de dependência maior. O App depende do ator Ui para exibir as informações na tela. Por sua vez, o Ui depende do Terminal para efetivamente desenhar na tela. Por fim, o Terminal depende do App para poder enviar eventos de teclado para que o App possa processar.

Nesse segundo caso, a solução adotada foi inverter o sentido de uma dependência. Ao invés do Terminal enviar eventos para o App, o App solicita eventos ao Terminal através de *polling*. O Terminal fica encarregado de manter uma fila armazenando os eventos até que o App os solicite.

A segunda solução não garantidamente é satisfatória universalmente. Portanto, se faz necessária a pesquisa e desenvolvimento de uma solução mais robusta para esse problema.

7.1.4 Supervisores

Nas implementações conhecidas do Modelo de Atores, em geral existe um tipo de ator especial chamado de supervisor. Esse ator é responsável por cuidar do ciclo de vida dos atores e suas dependências. O principal papel do supervisor é recriar atores que morreram inesperadamente. Em seguida, ele deve ser capaz de reinjetar os atores ressuscitados nos atores que dependem deles.

Usando a reescrita do Patch-Hub como exemplo, o ator App poderia ser elevado à categoria de supervisor. Caso um ator como o Net viesse a sofrer uma falha fatal, o supervisor poderia recriá-lo e reinjetá-lo no ator LoreApi, assim permitindo que o LoreApi e todo resto da aplicação continuem funcionando normalmente.

Portanto, a proposta apresentada poderia ser enriquecida com a adição de um ator supervisor, além de mecanismos para permitir a injeção de dependências após a inicialização.

Referências

[Hewitt 2010] Carl Hewitt. *Actor Model of Computation: Scalable Robust Information Systems*. Ago. de 2010. arXiv: 1008.1459. URL: https://arxiv.org/abs/1008.1459 (acesso em 20/11/2025) (citado na pg. 7).

[Kassab *et al.* 2018] Mohamad Kassab, Manuel Mazzara, JooYoung Lee e Giancarlo Succi. "Software architectural patterns in practice: an empirical study". *Innovations in Systems and Software Engineering* 14 (dez. de 2018), pp. 263–271. DOI: 10.1007/s11334-018-0319-4. URL: https://link.springer.com/article/10.1007/s11334-018-0319-4 (acesso em 20/11/2025) (citado na pg. 4).

[Kouraklis 2016] John Kouraklis. "Mvvm as design pattern". In: out. de 2016. ISBN: 978-1-4842-2213-3. DOI: 10.1007/978-1-4842-2214-0_1 (citado na pg. 5).

[KWorkflow 2025a] KWorkflow. *KWorkflow*. 2025. URL: https://kworkflow.org/ (acesso em 16/10/2025) (citado na pg. 32).

[KWorkflow 2025b] KWorkflow. *patch-hub*. 2025. URL: https://github.com/kworkflow/patch-hub (acesso em 16/10/2025) (citado na pg. 32).

[Linux Kernel Documentation 2025a] Linux Kernel Documentation. *Submitting patches*. 2025. URL: https://docs.kernel.org/process/submitting-patches.html (acesso em 16/10/2025) (citado na pg. 31).

[Linux Kernel Documentation 2025b] Linux Kernel Documentation. *Subsystem APIs*. 2025. URL: https://docs.kernel.org/subsystem-apis.html (acesso em 16/10/2025) (citado na pg. 31).

[Martin 2000] Robert C. Martin. *Design Principles and Design Patterns*. Rel. técn. Accessed: 2025-12-13. Object Mentor, 2000. URL: https://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf (citado na pg. 4).

[Reenskaug 1979] Trygve Reenskaug. *MODELS - VIEWS - CONTROLLERS*. Technical Note. Accessed: 2025-12-13. Xerox PARC, dez. de 1979. URL: https://jpaulgibson.synology.me/~jpaulgibson/TSP/Teaching/Teaching-ReadingMaterial/Reenskaug79b.pdf (citado na pg. 5).

[Rust Project Developers 2025] Rust Project Developers. *Rust: Uma linguagem empoderando todos a construir softwares confiáveis e eficientes*. 2025. URL: https://rust-lang.org/pt-BR/ (acesso em 20/11/2025) (citado na pg. 15).

[Somers 2025] Andrew Somers. *Mockall: A powerful mock object library for Rust*. 2025. URL: https://github.com/asomers/mockall (acesso em 20/11/2025) (citado na pg. 26).

[Tadokoro 2025] David Tadokoro. *The lore.kernel.org API*. Set. de 2025. URL: https://blog.kworkflow.org/the-lore.kernel.org-api/ (acesso em 16/10/2025) (citado na pg. 32).

[Tokio Contributors 2025] Tokio Contributors. *Tokio: Build reliable network applications without compromising speed*. 2025. URL: https://tokio.rs/ (acesso em 20/11/2025) (citado na pg. 19).

[Wikipedia 2025a] Wikipedia. *Linux*. 2025. URL: https://pt.wikipedia.org/wiki/Linux (acesso em 16/10/2025) (citado na pg. 31).

[Wikipedia 2025b] Wikipedia. *Linux (núcleo)*. 2025. URL: https://pt.wikipedia.org/wiki/Linux_(n%C3%BAcleo) (acesso em 16/10/2025) (citado na pg. 31).