# Relatório Detalhado: Arquitetura Orientada a Atores em Elixir para Simulação de Colisões Elásticas Concorrentes

Por: Matheus de Camargo Marques, um estudante aleatoriamente determinado a estudar sistemas quânticos de probabilidades infinitas.

## Resumo Executivo

Este relatório explora a aplicação da Arquitetura Orientada a Atores, implementada através do Elixir e do Erlang/OTP, para o desenvolvimento de sistemas concorrentes, com foco específico em uma simulação de colisões elásticas. A arquitetura de atores oferece uma abordagem robusta para gerenciar a complexidade da concorrência e da distribuição, aproveitando princípios como encapsulamento de estado, passagem assíncrona de mensagens e isolamento de falhas. O Elixir, com sua base na Máquina Virtual Erlang (BEAM), fornece um ambiente ideal para a materialização desses princípios por meio de processos leves e árvores de supervisão.

A simulação de colisões elásticas, que exige cálculos precisos e a coordenação de múltiplos objetos em movimento, beneficia-se intrinsecamente da capacidade do Modelo de Ator de lidar com um grande número de entidades independentes e interativas. A discussão abrange desde os fundamentos teóricos do Modelo de Ator até as ferramentas práticas do Elixir, como GenServer, Supervisor, Registry e ETS. Além disso, são abordadas técnicas de otimização espacial, como Hashing Espacial e Quadtrees, essenciais para gerenciar a detecção de colisões em ambientes com alta densidade de objetos. O relatório conclui que a combinação da Arquitetura Orientada a Atores com as capacidades do Elixir/OTP oferece uma solução escalável, tolerante a falhas e eficiente para simulações físicas complexas, mitigando desafios comuns de concorrência e depuração através de um design cuidadoso e da alavancagem das características da plataforma.

## 1. Introdução à Arquitetura Orientada a Atores

### 1.1. Conceitos Fundamentais do Modelo de Ator

O Modelo de Ator, um modelo matemático de computação concorrente concebido em 1973, postula que um ator é a unidade computacional mais básica, onde "tudo é um ator".[1] A interação entre atores ocorre exclusivamente por meio da passagem assíncrona de mensagens, o que constitui a base de sua operação.[2]

As características essenciais que definem o comportamento e a interação de um ator incluem:

- **Encapsulamento:** Cada ator mantém seu próprio estado privado e comportamento, garantindo que seu estado interno não seja diretamente acessível ou modificável por outros atores. Esse encapsulamento assegura mudanças de estado controladas e previsíveis, mitigando efetivamente as complexidades associadas ao estado mutável compartilhado em ambientes concorrentes.[2] A adesão a este princípio de "não compartilhamento de estado mutável" é uma escolha arquitetural deliberada que aborda os aspectos mais complexos e propensos a erros da programação concorrente, como condições de corrida,
*deadlocks* e os intrincados mecanismos de sincronização exigidos em modelos de memória compartilhada.[4] Ao isolar estritamente o estado dentro de atores individuais e impor a comunicação apenas por meio de passagem de mensagens, o Modelo de Ator simplifica inerentemente o raciocínio sobre o comportamento concorrente. Essa simplificação é o fator fundamental que leva a muitos dos benefícios centrais do modelo, como concorrência natural, escalabilidade mais fácil e redução da carga para o desenvolvedor no gerenciamento de primitivas de sincronização de baixo nível.[7] A ausência de estado mutável compartilhado significa que um desenvolvedor pode se concentrar na lógica dentro de um único ator sem se preocupar com modificações externas e concorrentes em seu estado. Para uma simulação complexa como colisões elásticas, onde inúmeras entidades (partículas) interagem e atualizam seus estados concorrentemente, este princípio é inestimável. Se cada partícula for modelada como um ator, ela gerencia sua própria posição, velocidade e massa. As atualizações dessas propriedades ocorrem de forma segura dentro do
*loop* de processamento sequencial do ator, eliminando a necessidade de bloqueios ou semáforos explícitos. Isso reduz significativamente a complexidade de implementação e depuração de atualizações concorrentes nos estados das partículas, permitindo uma simulação mais robusta e previsível.

- **Passagem de Mensagens Assíncrona:** Atores se comunicam enviando mensagens imutáveis a outros atores sem aguardar uma resposta imediata.[2] Este estilo de comunicação não bloqueante é crucial para permitir alta concorrência e desacoplamento de componentes. A natureza assíncrona da comunicação entre atores [2] implica um padrão de interação "disparar e esquecer".[3] Isso significa que, ao enviar uma mensagem, um ator não bloqueia ou espera por uma resposta; em vez disso, ele continua imediatamente seu próprio processamento. Esse mecanismo permite o *buffer* de mensagens em caixas de correio [5], permitindo que os remetentes prossigam com seu trabalho sem estarem acoplados à velocidade de processamento ou disponibilidade do destinatário. Esse comportamento inerente de não bloqueio é a causa principal do acoplamento fraco entre atores [6] e aumenta significativamente a resiliência do sistema. Se um ator destinatário estiver lento ou falhar temporariamente, o remetente não é diretamente impactado ou bloqueado, evitando falhas em cascata e permitindo que o sistema continue operando.[7] Em uma simulação de colisão elástica, isso significa que um ator de partícula, ao detectar uma colisão potencial, pode enviar uma mensagem de evento de colisão para um ator "resolutor de colisões" sem pausar seu próprio movimento ou esperar que a colisão seja totalmente resolvida. Isso permite que a simulação progrida continuamente, potencialmente processando outros movimentos de partículas ou detectando outras colisões em paralelo. Essa abordagem assíncrona é particularmente benéfica em um ambiente distribuído, onde a latência da rede poderia introduzir atrasos e gargalos significativos, garantindo maior *throughput* geral e responsividade da simulação.
- **Caixas de Correio (*Mailboxes*):** Cada ator é equipado com uma caixa de correio, ou fila de mensagens, onde as mensagens recebidas são armazenadas. As mensagens são processadas sequencialmente, geralmente na ordem Primeiro a Entrar, Primeiro a Sair (FIFO), o que simplifica o gerenciamento da concorrência dentro do ator e previne inerentemente problemas como condições de corrida no nível do estado interno do ator.[2] Um ator processa apenas uma mensagem por vez, garantindo a modificação segura do estado interno.[5]
- **Criação de Ator:** Atores possuem a capacidade de criar um número finito de novos atores dinamicamente. Esse mecanismo é vital para distribuir cargas de trabalho, gerenciar tarefas de forma eficiente e adaptar-se a demandas computacionais variáveis.[3] O ator criador frequentemente atribui um identificador único ou endereço ao novo ator para comunicação futura.[2]
- **Designação de Comportamento:** Um ator pode mudar dinamicamente seu comportamento (a lógica que aplica às mensagens recebidas) para mensagens subsequentes. Isso permite que os atores evoluam sua lógica de processamento com base nas mensagens recebidas ou no estado interno, oferecendo flexibilidade significativa.[4]
- **Terminação:** Atores podem ser encerrados de forma graciosa ao completar suas tarefas ou em resposta a erros irrecuperáveis, permitindo a limpeza de recursos e o desligamento ordenado.[4]

O Modelo de Ator difere fundamentalmente dos modelos computacionais tradicionais (por exemplo, Máquinas de Turing) que dependem de um estado global. Ele opera sem um estado global, abraçando o "não determinismo ilimitado", uma propriedade em que o tempo de atraso no atendimento de uma solicitação pode se tornar ilimitado devido à arbitragem de contenção por recursos compartilhados, mas ainda assim garante o serviço.[1] Isso contrasta com modelos que podem terminar em apenas um número limitado de estados, oferecendo uma abordagem mais robusta para interações assíncronas do mundo real.[1]

### 1.2. Vantagens para Sistemas Concorrentes e Distribuídos

O Modelo de Ator oferece vantagens convincentes para a construção de sistemas concorrentes e distribuídos robustos, escaláveis e tolerantes a falhas [2]:

- **Concorrência Natural e Escalabilidade:** Como os atores processam mensagens de forma independente e assíncrona, eles podem ser executados concorrentemente sem interferir uns com os outros.[4] Esse paralelismo inerente permite que os sistemas escalem horizontalmente, criando dinamicamente mais atores e distribuindo cargas de trabalho em múltiplos núcleos ou máquinas.[4] O Elixir, aproveitando a VM Erlang, é especificamente projetado para lidar com milhões de conexões concorrentes de forma eficiente.[4] As várias vantagens do Modelo de Ator — concorrência natural, escalabilidade, tolerância a falhas e acoplamento fraco — não são características independentes, mas sim consequências profundamente interconectadas de seus princípios centrais. O isolamento do estado de um ator [2] combinado com sua dependência da passagem de mensagens assíncronas [2] é a causa primária que permite a concorrência natural.[4] Isso, por sua vez, constitui o mecanismo fundamental para alcançar a escalabilidade horizontal [4], à medida que os atores podem ser distribuídos em múltiplas unidades computacionais (núcleos, máquinas) sem encontrar conflitos de memória compartilhada. A filosofia "deixe falhar" [8], um pilar do OTP, torna-se uma estratégia de tolerância a falhas viável e eficaz precisamente por causa desse isolamento inerente; uma falha dentro de um ator não corrompe um estado global compartilhado, permitindo assim a recuperação e reinício localizados sem impacto generalizado no sistema. Para uma simulação de colisão em larga escala, isso significa que cada partícula individual pode ser efetivamente modelada como um ator. À medida que o número de partículas na simulação aumenta, o sistema pode ser escalado simplesmente adicionando mais recursos computacionais (por exemplo, mais núcleos de CPU ou máquinas adicionais) e distribuindo os atores de partículas por esses recursos. A simulação permanece altamente resiliente porque, se qualquer ator de partícula individual falhar (por exemplo, devido a um erro de precisão numérica ou um estado inesperado), seu supervisor pode reiniciá-lo automaticamente, potencialmente reinicializando seu estado a partir da última configuração conhecida e válida. Isso impede que uma única entidade defeituosa cause uma falha catastrófica em toda a simulação.
- **Tolerância a Falhas e Resiliência:** Atores são unidades isoladas de computação, cada uma com seu próprio estado.[2] Esse isolamento impede que as falhas se propaguem, contendo-as dentro de atores individuais e prevenindo falhas em todo o sistema.[5] Isso é ainda mais aprimorado por estratégias hierárquicas de supervisão, onde atores pais monitoram e reiniciam automaticamente atores filhos, levando a "sistemas auto-recuperáveis" que podem se recuperar graciosamente de erros.[2]
- **Gerenciamento Simplificado da Concorrência:** Ao abstrair a concorrência em atores autocontidos e depender da passagem de mensagens, o Modelo de Ator simplifica significativamente o desenvolvimento de aplicações concorrentes. Ele elimina a necessidade de os desenvolvedores gerenciarem problemas de sincronização de baixo nível, bloqueios ou semáforos.[4]
- **Acoplamento Fraco:** A comunicação através de mensagens assíncronas desacopla inerentemente o remetente do receptor. Isso significa que o produtor da mensagem não precisa de conhecimento explícito de quantos assinantes existem ou como alcançá-los [9], o que aprimora a modularidade, flexibilidade e evolução independente dos componentes.[6]
- **Transparência de Localização:** O Modelo de Ator abstrai a localização física dos atores. Atores podem ser distribuídos de forma transparente em diferentes máquinas ou nós sem exigir alterações no design geral do sistema ou na lógica de comunicação, pois as interações são baseadas em endereços de atores.[2] A capacidade do Modelo de Ator de cumprir suas promessas de robustez, escalabilidade e desempenho no Elixir depende profundamente da Máquina Virtual Erlang (BEAM). A arquitetura única da BEAM, caracterizada por seus processos leves, mecanismos altamente eficientes de passagem de mensagens e suporte integrado para concorrência e distribuição [4], é o que traduz as vantagens teóricas do Modelo de Ator em capacidades tangíveis e de alto desempenho em aplicações Elixir.[12] Sem o *runtime* maduro e testado em batalha da BEAM, as complexidades inerentes ao gerenciamento de milhões de operações concorrentes e comunicação distribuída seriam significativamente mais difíceis de superar, e os benefícios teóricos seriam muito mais desafiadores de alcançar com tal confiabilidade e desempenho. Isso destaca que a decisão de escolher o Elixir para uma simulação orientada a atores não se trata apenas da sintaxe da linguagem ou dos paradigmas de programação funcional. Trata-se fundamentalmente de alavancar um ambiente de *runtime* maduro e construído para um propósito específico (a BEAM) que foi projetado ao longo de décadas especificamente para construir sistemas altamente concorrentes, distribuídos e tolerantes a falhas. Essa plataforma subjacente é um componente arquitetural crítico que influencia diretamente a confiabilidade, o perfil de desempenho e a facilidade de desenvolvimento de todo o sistema para uma simulação complexa.

### 1.3. Desafios e Considerações Práticas

Apesar de suas vantagens, o Modelo de Ator apresenta certos desafios práticos e considerações para os desenvolvedores [2]:

- **Complexidade de Depuração:** A depuração de sistemas distribuídos construídos com passagem de mensagens assíncrona pode ser inerentemente desafiadora devido à natureza não determinística da entrega de mensagens e às interações complexas e entrelaçadas entre os atores.[3] Ferramentas de depuração tradicionais podem ter dificuldade em fornecer um rastreamento claro e linear da execução. Ferramentas de rastreamento eficazes e registro abrangente são essenciais para entender o comportamento do sistema.[7] A natureza assíncrona da comunicação entre atores [2], embora seja um poderoso facilitador para escalabilidade e acoplamento fraco, introduz simultaneamente desafios significativos na depuração.[3] A ausência de chamadas de método diretas e bloqueantes e o tempo não determinístico do processamento de mensagens em múltiplos atores tornam difícil reconstruir a sequência exata de eventos que levaram a um determinado estado ou erro. Isso representa uma troca fundamental: ganhos em desempenho e resiliência geralmente vêm ao custo de observabilidade imediata e depuração direta. Isso implica que um conjunto diferente de estratégias de depuração, focado na correlação de eventos e rastreamento em todo o sistema, é necessário. Para uma simulação de colisão complexa, se uma partícula exibe um comportamento inesperado (por exemplo, velocidade incorreta após uma colisão, ou um objeto passando por outro), rastrear a sequência exata de mensagens, mudanças de estado e interações entre potencialmente centenas ou milhares de atores interagentes pode ser extremamente complexo. Isso exige a implementação de registro robusto, ferramentas de monitoramento avançadas (como o
observer do Erlang), e potencialmente padrões de *event sourcing* ou mecanismos de rastreamento personalizados para permitir uma análise *post-mortem* eficaz e a compreensão do comportamento dinâmico da simulação.
- **Garantias de Ordem de Mensagens:** Embora atores individuais processem mensagens de sua caixa de correio em ordem FIFO [2], garantir a ordem precisa das mensagens em múltiplos atores ou lidar com cenários de perda ou duplicação de mensagens em ambientes altamente distribuídos requer design cuidadoso e mecanismos adicionais.[3]
- **Potencial para *Deadlocks* e Transbordamento de Caixas de Correio:** Embora problemas de concorrência relacionados ao estado compartilhado sejam mitigados, os atores ainda podem ser suscetíveis a *deadlocks* lógicos se as dependências de mensagens criarem condições de espera circulares entre os atores.[2] Além disso, as caixas de correio podem transbordar se as mensagens forem produzidas a uma taxa significativamente mais rápida do que podem ser consumidas pelo ator receptor, levando ao esgotamento de recursos.[2]
- **Granularidade do Ator:** Determinar a granularidade ideal para os atores é uma decisão de design crítica que impacta diretamente o desempenho. Muitos atores de granularidade fina podem introduzir uma sobrecarga excessiva de comunicação devido à passagem de mensagens, enquanto poucos atores de granularidade grossa (grandes) podem levar à contenção e uso ineficiente de recursos, anulando alguns dos benefícios da concorrência.[7] O texto destaca explicitamente que "a granularidade dos atores afeta o desempenho".[7] Este é um ponto de decisão arquitetural crítico. Se, por exemplo, cada cálculo de baixo nível dentro de uma simulação fosse tratado por um ator separado, a sobrecarga da passagem de mensagens entre esses atores de granularidade fina poderia se tornar proibitiva, anulando os benefícios da concorrência. Por outro lado, se os atores forem muito grandes (por exemplo, um único ator gerenciando uma região de simulação grande inteira), isso poderia levar a contenção interna e limitar o potencial de execução paralela. Esta observação revela uma clara relação causal entre a escolha de design da granularidade do ator e o desempenho e eficiência resultantes do sistema. O desempenho ótimo exige um equilíbrio cuidadoso. Em nossa simulação de colisão elástica, isso significa que a decisão sobre o que constitui um "ator" é primordial. Cada partícula individual deve ser um ator? Ou os atores devem gerenciar grupos de partículas dentro de uma região espacial? Ou deve haver atores dedicados para fases computacionais específicas, como detecção de colisão? A escolha influenciará profundamente o volume de tráfego de mensagens, o potencial de gargalos e a eficiência geral das fases de detecção e resolução de colisões. Essa consideração aponta diretamente para a necessidade de empregar técnicas de particionamento espacial (discutidas na Seção 5) como uma solução estratégica para gerenciar e otimizar efetivamente a granularidade do ator, reduzindo a comunicação desnecessária entre atores.
- **Sobrecarga de Criação de Ator:** Embora os processos Elixir sejam leves, a criação dinâmica de um número extremamente grande de atores de vida muito curta ainda pode introduzir alguma sobrecarga, o que precisa ser considerado em cenários de alto *throughput*.[7]
- **Aplicabilidade:** O Modelo de Ator pode nem sempre ser o paradigma mais adequado. Para problemas puramente síncronos ou aqueles que não se beneficiam inerentemente de alta concorrência e distribuição, a introdução do Modelo de Ator pode adicionar sobrecarga e complexidade desnecessárias.[3]

A Tabela 1 oferece uma comparação concisa das vantagens e desvantagens do Modelo de Ator, consolidando informações cruciais para a tomada de decisões arquiteturais.

**Tabela 1: Comparativo de Vantagens e Desvantagens do Modelo de Ator**

| Vantagens (Pros) | Desvantagens (Cons) |
|---|---|
| Facilidade de Escala [2] | Suscetibilidade a Deadlocks [2] |
| Tolerância a Falhas [2] | Mailboxes Transbordando [2] |
| Distribuição Geográfica [2] | Complexidade de Depuração [3] |
| Não Compartilhamento de Estado [2] | Ordem de Mensagens [3] |
| Concorrência Simplificada [4] | Overhead de Criação de Ator [7] |
| Acoplamento Fraco [6] | Granularidade do Ator [7] |
| Transparência de Localização [2] | Não Aplicável a Problemas Síncronos [3] |
| Sistemas Auto-curáveis [2] | |

## 2. O Modelo de Ator em Elixir e OTP

### 2.1. Processos Leves e Passagem de Mensagens

O Elixir, construído sobre a robusta Máquina Virtual Erlang (BEAM), suporta fundamentalmente os princípios centrais do Modelo de Ator. Embora o termo "ator" como formalmente definido possa não se aplicar diretamente a todo processo Elixir, os processos subjacentes e os mecanismos de passagem de mensagens são altamente semelhantes a uma abordagem baseada em atores.[12] O Elixir alavanca esses processos leves e a passagem eficiente de mensagens como seus primitivos fundamentais para a concorrência.[4]

Esses processos são caracterizados por seu isolamento, independência e comunicação exclusiva por meio de passagem assíncrona de mensagens, incorporando os princípios essenciais do Modelo de Ator.[4] Eles são excepcionalmente leves, permitindo que o sistema lide com milhões de conexões concorrentes com sobrecarga mínima de recursos.[4] As mensagens, uma vez enviadas, são colocadas na caixa de correio do destinatário e processadas sequencialmente, o que inerentemente simplifica o gerenciamento da concorrência e previne condições de corrida dentro do estado do processo.[4] A distinção crucial é que, embora os processos Elixir exibam os comportamentos chave dos atores (estado isolado, passagem de mensagens assíncronas, processamento sequencial de mensagens), eles podem não aderir estritamente a cada axioma do Modelo de Ator matemático.[1] Em vez disso, eles representam uma realização altamente otimizada, pragmática e testada em batalha da concorrência tipo ator dentro da VM BEAM. Essa distinção é vital para um público tecnicamente astuto, pois enquadra a abordagem do Elixir como uma solução prática e de alto desempenho, em vez de um teste de pureza teórica. Isso significa que os desenvolvedores devem priorizar a compreensão e o aproveitamento dos benefícios práticos e dos comportamentos idiomáticos do OTP (como

GenServer) que o Elixir oferece, em vez de se prenderem às nuances da pureza teórica. A BEAM fornece garantias robustas para tolerância a falhas, concorrência e distribuição, mesmo que seus processos sejam "como atores" em vez de uma implementação direta e formal de um Modelo de Ator puro. Essa compreensão orienta as decisões arquiteturais em direção a soluções pragmáticas e de alto desempenho.

**2.2. GenServer: O Ator Fundamental em Elixir**

O comportamento GenServer no Elixir serve como a abstração idiomática e de alto nível para implementar processos semelhantes a atores.[12] Ele fornece uma maneira padronizada de construir abstrações cliente-servidor, gerenciar estado interno e lidar com mensagens síncronas (

handle_call/2) e assíncronas (handle_cast/2).[13] Um

GenServer mantém seu estado em um *loop* contínuo de execução [12] e processa as mensagens recebidas uma a uma, garantindo a consistência interna.

Os *callbacks* chave que definem o comportamento de um GenServer incluem [14]:

- init/1: Esta função é invocada quando o processo GenServer é iniciado, sendo responsável por definir seu estado inicial. É importante notar que operações de longa duração dentro de init/1 podem bloquear a inicialização do processo, sugerindo a necessidade de inicialização assíncrona, como o envio de uma mensagem para si mesmo para disparar etapas adicionais de inicialização de forma assíncrona.[14]
- handle_call/2: Lida com mensagens síncronas enviadas ao GenServer usando a função call. Ele processa a mensagem, pode atualizar o estado do GenServer e envia uma resposta de volta ao chamador.[14]
- handle_cast/2: Recebe mensagens enviadas assincronamente ao GenServer usando a função cast. Ele processa a mensagem e pode atualizar o estado do GenServer sem enviar uma resposta.[14]
- handle_info/2: Invocado sempre que o GenServer recebe uma mensagem que não corresponde a nenhum dos padrões de mensagem síncronos ou assíncronos definidos (tratados por handle_call/2 e handle_cast/2, respectivamente).[14]

A flexibilidade do GenServer permite que seu código seja alterado em tempo de execução sem interromper o processo, uma característica poderosa para sistemas de alta disponibilidade.[14]

### 2.3. Árvores de Supervisão e Tolerância a Falhas

As Árvores de Supervisão são um conceito fundamental no OTP, projetado para gerenciar falhas de forma graciosa e garantir que as aplicações possam se recuperar de erros sem consequências catastróficas.[8] Elas formam uma estrutura hierárquica onde cada nó representa um processo, e esses processos são organizados em uma topologia de árvore.[8] A filosofia subjacente das Árvores de Supervisão é "deixe falhar".[8] Isso significa que, em vez de tentar evitar todas as falhas, o sistema é projetado para reiniciar automaticamente um processo que falhou por seu supervisor, garantindo que o sistema retorne a um estado conhecido e continue a funcionar conforme o esperado.[8]

A anatomia de uma Árvore de Supervisão consiste em:

- **Supervisores:** São processos responsáveis por gerenciar outros processos e atuam como os nós internos da Árvore de Supervisão.[8] Os supervisores definem as estratégias para reiniciar seus processos filhos quando ocorre uma falha.[8] No Elixir, os supervisores são implementados usando o comportamento Supervisor.[13]
- **Trabalhadores (*Workers*):** São os nós folha da Árvore de Supervisão e são os processos reais que realizam as tarefas da aplicação.[8] Os trabalhadores podem ser qualquer processo Elixir, como GenServers, Tasks ou qualquer processo personalizado que cumpra uma função específica.[8]
- **Filhos (*Children*):** Este termo se refere tanto a supervisores quanto a trabalhadores dentro da Árvore de Supervisão. Cada supervisor é um filho de seu supervisor pai, estabelecendo a estrutura hierárquica.[8]

Os supervisores empregam várias **Estratégias de Reinício** para lidar com falhas de processos filhos [8]:

- **One-for-One (:one_for_one):** Se um processo filho falhar, apenas esse filho específico é reiniciado, deixando os outros filhos inalterados. Esta estratégia é adequada quando os filhos são independentes e suas falhas não afetam uns aos outros.[8]
- **One-for-All (:one_for_all):** Se qualquer processo filho falhar, todos os outros filhos na árvore são terminados e então reiniciados juntos. Isso é útil quando os filhos são interdependentes e o sistema precisa estar em um estado consistente.[8]
- **Rest-for-One (:rest_for_one):** Quando um processo filho falha, todos os processos filhos que foram iniciados *depois* dele são terminados, e então todos os filhos terminados (incluindo o que falhou) são reiniciados.[8]
- **Rest-for-All (:rest_for_all):** Se um processo filho falhar, todos os processos filhos são terminados e então reiniciados juntos. Esta estratégia é raramente usada.[8]

As Árvores de Supervisão também podem ser **Supervisores Dinâmicos**, permitindo a adição e remoção de processos filhos em tempo de execução, o que proporciona flexibilidade para requisitos em constante mudança.[15]

### 2.4. Registro de Processos com Registry

O módulo Registry do Elixir oferece um mecanismo para gerenciar grupos de processos dinâmicos, permitindo que os processos se registrem sob nomes únicos, o que facilita a busca e a interação.[17] Isso é particularmente útil em sistemas distribuídos e aplicações tolerantes a falhas onde os processos precisam ser agrupados e gerenciados dinamicamente.[17]

A utilização do Registry envolve as seguintes etapas:

1. **Configuração de um Registro:** O primeiro passo é definir e iniciar um processo Registry, tipicamente como parte da árvore de supervisão da sua aplicação.[17] Um exemplo de configuração em MyApp.Application seria {Registry, keys: :unique, name: MyApp.Registry}, onde keys: :unique garante que cada nome registrado seja único e name: MyApp.Registry atribui um nome global ao processo Registry.[17]
2. **Registro de Processos:** Uma vez configurado o Registry, processos individuais podem se registrar sob nomes únicos. Este registro permite que outros processos os encontrem e se comuniquem com eles.[17] Existem duas abordagens principais para registrar processos:
    - **{:via, Registry, {...}}:** Esta é uma sintaxe especial usada principalmente para iniciar ou referenciar um processo com um registro. Ela sinaliza ao Elixir que se deseja buscar ou registrar um processo através de um registro personalizado e associá-lo a uma chave. É tipicamente usado como o argumento name ao iniciar um processo com funções como GenServer.start_link/3.[18] A vantagem é o registro automático na inicialização do processo, eliminando a necessidade de uma chamada de registro separada.[18]
    - **Registry.register/3:** Esta é uma chamada de função direta usada para registrar manualmente o processo atual com um registro. É invocada dentro do processo que se deseja registrar, geralmente logo após o processo ter sido iniciado. Este método oferece maior controle sobre quando e onde o registro ocorre e permite a inclusão de metadados adicionais durante o registro.[18]
3. **Busca de Processos Registrados:** Processos podem ser buscados usando os nomes pelos quais foram registrados. Esse mecanismo de busca é crucial para a interação dinâmica com os processos.[17] A função Registry.lookup/2 é utilizada para encontrar o ID do processo (PID) associado a um determinado nome em uma instância específica do Registry. Uma vez obtido o PID, outros processos podem então enviar mensagens ou fazer requisições GenServer.call para o processo encontrado.[17]

Em essência, o Registry do Elixir atua como um diretório central onde os processos podem anunciar sua presença com um identificador único (um nome), e outros processos podem então consultar este diretório para encontrar os PIDs desses processos registrados. Essa abstração simplifica o gerenciamento de processos, especialmente em ambientes concorrentes e distribuídos, desacoplando a identidade do processo de seus PIDs voláteis.[17]

**2.5. Armazenamento em Memória com ETS**

Erlang Term Storage (ETS) é um poderoso motor de armazenamento em memória embutido no OTP, disponível para uso no Elixir para armazenamento de dados em memória.[19] O ETS é projetado para armazenar grandes volumes de dados e oferece acesso aos dados em tempo constante, O(1).[19]

Suas características chave e uso incluem:

- **Criação e Propriedade da Tabela:** As tabelas ETS são criadas usando a função :ets.new/2, que aceita um nome de tabela e um conjunto de opções. Cada tabela é de propriedade do processo que a cria. Quando o processo proprietário termina, todas as suas tabelas ETS associadas são automaticamente destruídas.[19] É possível ter um número ilimitado de tabelas ETS, limitado apenas pela memória do servidor, configurável pela variável de ambiente ERL_MAX_ETS_TABLES.[19] As tabelas podem ser acessadas por um identificador retornado ou por nome se a opção :named_table for incluída durante a criação.[19]
- **Tipos de Tabela:** O ETS oferece quatro tipos distintos de tabela, cada um com comportamentos específicos [19]:
  - set: O tipo padrão. Impõe chaves únicas, permitindo apenas um valor por chave.
  - ordered_set: Semelhante a set, mas os dados são ordenados por termos Erlang/Elixir. A comparação de chaves é baseada na igualdade (ex: 1 e 1.0 são considerados iguais).
  - bag: Permite múltiplos objetos por chave, mas apenas uma instância de cada objeto único por chave.
  - duplicate_bag: Permite múltiplos objetos por chave, incluindo instâncias duplicadas do mesmo objeto.
- **Controles de Acesso:** As tabelas ETS possuem controle de acesso semelhante ao controle de acesso de módulos [19]:
  - public: Permite acesso de leitura e escrita a todos os processos.
  - protected: Permite acesso de leitura a todos os processos, mas apenas o processo proprietário pode escrever na tabela. Este é o controle de acesso padrão.
  - private: Restringe o acesso de leitura e escrita exclusivamente ao processo proprietário.
- **Inserção de Dados:** As tabelas ETS não possuem esquema, sendo a única limitação que os dados devem ser armazenados como uma tupla cujo primeiro elemento é a chave.[19] Novos dados são adicionados usando :ets.insert/2. Para tabelas set e ordered_set, insert/2 substituirá dados existentes se a chave já existir.[19] Para evitar a sobrescrita, pode-se usar :ets.insert_new/2, que retorna false se a chave já existir.
- **Recuperação de Dados:** O ETS oferece várias maneiras convenientes e flexíveis de recuperar dados armazenados [19]:
  - **Busca por Chave (:ets.lookup/2):** O método mais eficiente, recuperando todos os registros associados a uma dada chave.
  - **Correspondências Simples (:ets.match/2 e :ets.match_object/2):** Permitem correspondência de padrões usando variáveis (ex: :"$1") e o curinga :_.
  - **Busca Avançada (:ets.select/2 e :ets.fun2ms/1):** Para consultas mais complexas, :ets.select/2 é usado com uma lista de tuplas de 3 aridades. :ets.fun2ms/1 simplifica a criação de especificações de correspondência, permitindo que as consultas sejam escritas com uma sintaxe de função mais familiar.[19]
- **Exclusão de Dados:**
  - **Remoção de Registros (:ets.delete/2):** Exclui um registro específico (chave e seus valores) da tabela.
  - **Remoção de Tabelas (:ets.delete/1):** Permite a exclusão de uma tabela ETS inteira sem terminar o processo proprietário, já que as tabelas ETS não são coletadas como lixo de outra forma.[19]

O ETS é uma solução de armazenamento em memória altamente eficiente e flexível no Elixir, crucial para aplicações que exigem acesso rápido a dados e capazes de lidar com grandes conjuntos de dados. Seu modelo de propriedade de processo, vários tipos de tabela e mecanismos de recuperação robustos o tornam uma ferramenta fundamental para a construção de aplicações Elixir de alto desempenho.[19]

### 2.6. Distribuição e Clustering de Nós

A capacidade do Elixir de operar em sistemas distribuídos é uma de suas maiores forças, herdada diretamente da Máquina Virtual Erlang (BEAM). Um sistema Erlang distribuído consiste em vários sistemas de *runtime* Erlang que se comunicam entre si, sendo cada um desses sistemas chamado de "nó".[11] Para que os nós se comuniquem, eles precisam ser nomeados e compartilhar um "cookie" secreto.[21]

A comunicação entre nós é transparente para os atores, que interagem por meio de endereços, independentemente de estarem local ou remotamente.[2] Isso é facilitado por bibliotecas como

dns_cluster, que simplifica a configuração de *clustering* usando DNS, especialmente em ambientes como Fly.io.[21] O

dns_cluster permite que os nós descubram uns aos outros e formem um *cluster* automaticamente.[21]

Para configurar o *clustering*:

1. **Nomear os Nós:** Os nós Elixir devem ser nomeados usando o nome da aplicação e o endereço IPv6 atribuído ao nó, o que facilita a formação do *cluster* via DNS.[21]
2. **Instalar e Usar dns_cluster:** A biblioteca dns_cluster é adicionada à aplicação, configurando-a para buscar outros nós implantados na rede privada (por exemplo, myapp.internal).[21]
3. **Executar Múltiplas VMs:** Para que o *clustering* ocorra, é necessário ter mais de uma máquina virtual (VM) executando a aplicação. Isso pode ser feito escalando a aplicação para ter múltiplas VMs em uma única região ou adicionando VMs em diferentes regiões para proximidade com os usuários.[21]
4. **Configurar o Cookie:** Antes que dois nós Elixir possam se agrupar, eles devem compartilhar um cookie secreto. Este cookie não é uma chave de criptografia, mas um mecanismo para criar múltiplos conjuntos de pequenos *clusters* na mesma rede que não se conectam todos juntos. O cookie pode ser definido através da variável de ambiente RELEASE_COOKIE.[21]

Embora o dns_cluster seja uma solução simples para *clustering* básico, para casos de uso mais avançados que exigem múltiplas estratégias de descoberta de nós ou reconexão automática, a biblioteca libcluster é geralmente recomendada por ser mais extensível e rica em recursos.[22]

## 3. Simulação de Colisões Elásticas

### 3.1. Princípios Físicos das Colisões Elásticas

Uma colisão elástica é um tipo de colisão em que a energia cinética total do sistema é conservada antes e depois do impacto.[23] Além da energia cinética, o momento linear total do sistema também é conservado.[23] Isso significa que, em um sistema isolado (onde não há forças externas atuando ou as forças externas se anulam), a soma dos momentos lineares de todas as partículas antes da colisão é igual à soma após a colisão, e o mesmo se aplica à energia cinética.[23]

Matematicamente, para uma colisão elástica entre duas partículas (massa m1, velocidade inicial v1i e massa m2, velocidade inicial v2i), as velocidades finais (v1f e v2f) podem ser determinadas pelas seguintes equações, que derivam da conservação do momento linear e da energia cinética [23]:

v1f=m1+m2(m1−m2)v1i+2m2v2i

v2f=m1+m22m1v1i+(m2−m1)v2i

$$v_{1f} = \frac{(m_1 - m_2)v_{1i} + 2m_2 v_{2i}}{m_1 + m_2}$$

$$v_{2f} = \frac{2m_1 v_{1i} + (m_2 - m_1)v_{2i}}{m_1 + m_2}$$

Um conceito importante em colisões é o centro de massa do sistema. Em um sistema isolado, o centro de massa continua a se mover com a mesma velocidade antes e depois da colisão, o que pode ser visualizado através de "fotos" sucessivas do centro de massa em intervalos de tempo iguais.[23] Se as duas partículas tiverem massas iguais (m1=m2), elas simplesmente trocam suas velocidades após a colisão.[23]

### 3.2. Desafios na Simulação Concorrente

A simulação de colisões elásticas em um ambiente concorrente apresenta vários desafios significativos:

- **Gerenciamento de Estado Compartilhado:** Em uma simulação, as posições e velocidades das partículas mudam continuamente. Se múltiplos processos tentarem atualizar o estado da mesma partícula ou de partículas interdependentes simultaneamente, podem ocorrer condições de corrida e inconsistências de dados, exigindo mecanismos complexos de bloqueio e sincronização.[4]
- **Detecção de Colisões:** Identificar quando e onde as colisões ocorrem entre um grande número de objetos é computacionalmente intensivo. Um método ingênuo de verificar cada par de objetos (O(N^2)) torna-se inviável para grandes simulações. A detecção eficiente de colisões é crucial para o desempenho.[24]
- **Resolução de Colisões:** Após a detecção, a resolução da colisão (ou seja, o cálculo das novas velocidades e, possivelmente, posições) deve ser precisa e atômica para evitar que os objetos se sobreponham ou se comportem de forma irrealista.
- **Escalabilidade:** À medida que o número de partículas aumenta, a demanda computacional cresce exponencialmente. O sistema deve ser capaz de escalar horizontalmente para distribuir a carga de trabalho em múltiplos núcleos ou máquinas.[4]
- **Sincronização de Tempo:** Em uma simulação baseada em tempo discreto, garantir que todos os atores progridam no tempo de forma consistente e que os eventos de colisão sejam processados na ordem correta é um desafio.

A arquitetura orientada a atores, com seu modelo de "não compartilhamento de estado" e comunicação assíncrona, oferece um paradigma promissor para abordar muitos desses desafios, permitindo que cada partícula ou região da simulação seja gerenciada por um ator isolado.

## 4. Arquitetura Orientada a Atores para Simulação de Colisões

### 4.1. Modelagem de Entidades como Atores

Na simulação de colisões elásticas, as entidades primárias são as partículas ou objetos. A modelagem dessas entidades como atores é uma aplicação natural dos princípios do Modelo de Ator. Cada partícula pode ser representada por um GenServer [12], que encapsula seu estado (massa, posição, velocidade) e comportamento (movimento, resposta a colisões).[4]

A arquitetura proposta seria:

- **Ator de Partícula (Particle.GenServer):**
  - **Estado:** %{mass: m, position: {x, y, z}, velocity: {vx, vy, vz}}.
  - **Comportamento:**
    - Recebe mensagens para atualizar sua posição e velocidade com base no tempo decorrido.
    - Recebe mensagens de "potencial colisão" de um ator de detecção de colisões.
    - Envia mensagens de "colisão confirmada" para um ator de resolução de colisões quando uma colisão real é detectada.
    - Atualiza seu próprio estado de forma segura e sequencial ao processar mensagens em sua caixa de correio.[4]
- **Ator de Detecção de Colisões (CollisionDetector.GenServer):**
  - Este ator seria responsável por identificar pares de partículas que estão próximas o suficiente para uma colisão potencial. Em vez de uma verificação O(N^2), ele utilizaria técnicas de particionamento espacial (discutidas na Seção 5) para otimizar a busca.[24]
  - **Comportamento:**
    - Recebe periodicamente as posições atualizadas de todas as partículas (ou de partículas em sua região atribuída).
    - Utiliza uma estrutura de dados espacial (como Quadtree ou Hashing Espacial) para agrupar partículas em regiões.
    - Identifica pares de partículas dentro da mesma região ou regiões adjacentes que podem colidir.
    - Envia mensagens de "potencial colisão" para os atores de partículas envolvidos, ou para um ator de resolução de colisões, contendo informações sobre os objetos e o tempo de colisão previsto.
- **Ator de Resolução de Colisões (CollisionResolver.GenServer):**
  - Este ator receberia as mensagens de colisão confirmada e aplicaria as leis de conservação do momento e da energia cinética para calcular as novas velocidades das partículas envolvidas.[23]
  - **Comportamento:**
    - Recebe mensagens de "colisão confirmada" (ou "resolver colisão") contendo os estados das duas partículas colidindo e o tempo da colisão.
    - Calcula as novas velocidades das partículas usando as equações de colisão elástica.[23]
    - Envia mensagens de "atualizar estado" de volta para os atores de partículas afetados com suas novas velocidades.
    - Pode usar um Registry para buscar os PIDs dos atores de partículas envolvidos.[17]
- **Ator de Gerenciamento da Simulação (SimulationManager.GenServer):**
  - Atua como o orquestrador principal da simulação.
  - **Comportamento:**
    - Inicia e supervisiona todos os outros atores (partículas, detectores, resolvedores).[8]
    - Controla o avanço do tempo na simulação, enviando mensagens de "passo de tempo" para os atores de partículas.
    - Gerencia o estado global da simulação (por exemplo, o número total de partículas, os limites do ambiente).
    - Pode coletar estatísticas da simulação.

A utilização de GenServer para cada um desses papéis permite que os atores mantenham seu próprio estado isolado e se comuniquem por meio de mensagens assíncronas, mitigando os problemas de concorrência de estado compartilhado.[4]

**4.2. Fluxo de Interação e Gerenciamento de Concorrência**

O fluxo de interação na simulação seria baseado em um modelo de eventos e mensagens assíncronas, aproveitando a capacidade do Elixir de lidar com alta concorrência.

1. **Inicialização:** O SimulationManager inicia uma árvore de supervisão que inclui o CollisionDetector, o CollisionResolver e um Supervisor para os Particle.GenServers (possivelmente um DynamicSupervisor para adicionar/remover partículas dinamicamente).[8] Cada
   Particle.GenServer é iniciado com seu estado inicial (massa, posição, velocidade).[14]
2. **Ciclo de Simulação (Passo de Tempo):**
   - O SimulationManager envia uma mensagem de "passo de tempo" para todos os Particle.GenServers.
   - Cada Particle.GenServer atualiza sua posição com base em sua velocidade e no tempo decorrido.
   - Os Particle.GenServers enviam suas posições atualizadas para o CollisionDetector. Isso pode ser feito via Registry para lookup eficiente ou via PubSub para *broadcast* de atualizações de posição.[10]
3. **Detecção de Colisões:**
   - O CollisionDetector recebe as posições das partículas.
   - Utilizando sua estrutura de particionamento espacial, ele identifica pares de partículas que estão em proximidade e podem colidir.
   - Para cada par potencial, o CollisionDetector envia uma mensagem de "potencial colisão" para o CollisionResolver, contendo os PIDs das duas partículas e seus estados relevantes.
4. **Resolução de Colisões:**
   - O CollisionResolver recebe as mensagens de "potencial colisão".
   - Ele realiza um teste de colisão mais preciso para confirmar se uma colisão realmente ocorreu dentro do passo de tempo.
   - Se uma colisão for confirmada, o CollisionResolver calcula as novas velocidades para as duas partículas envolvidas usando as equações de colisão elástica.[23]
   - O CollisionResolver então envia mensagens de "atualizar velocidade" para os Particle.GenServers afetados, que atualizam seus estados internos.
5. **Tolerância a Falhas:** Se qualquer Particle.GenServer ou outro ator falhar, seu supervisor o reiniciará automaticamente.[8] O estado do ator reiniciado pode ser restaurado a partir de um ponto de verificação ou re-inicializado, dependendo da estratégia de recuperação. A natureza isolada dos atores garante que a falha de um ator não derrube todo o sistema.[4]

A passagem de mensagens assíncronas entre atores garante que a simulação possa continuar a progredir mesmo enquanto colisões estão sendo detectadas e resolvidas em paralelo. Isso evita gargalos e permite que o sistema escale para um grande número de partículas.

#### 4.3. Gerenciamento de Estado e Otimização de Dados

O gerenciamento de estado em uma simulação de colisões é crucial para o desempenho e a consistência. Cada Particle.GenServer mantém seu próprio estado de forma encapsulada.[4] Para otimizar o acesso e a manipulação de dados em memória, o Erlang Term Storage (ETS) pode ser empregado.

- **ETS para Dados de Partículas:**
  - Embora cada Particle.GenServer mantenha seu estado, um ETS de tipo set ou ordered_set pode ser usado para armazenar uma cópia de referência das propriedades das partículas (por exemplo, massa, raio) que raramente mudam, ou para um cache de dados frequentemente acessados.[19] Isso permitiria que outros atores (como o CollisionDetector) acessassem rapidamente as propriedades das partículas sem precisar enviar mensagens síncronas de GenServer.call para cada partícula, reduzindo a sobrecarga de comunicação.
  - O CollisionDetector ou o SimulationManager poderiam ser os proprietários da tabela ETS, controlando a escrita (protected ou private access) e permitindo que outros atores leiam (public access).[19]
  - Para o *Collision Detector*, o ETS poderia armazenar a última posição conhecida de cada partícula, permitindo que o detector compare rapidamente as posições sem interrogar cada ator de partícula em cada ciclo. As atualizações seriam feitas por mensagens assíncronas dos atores de partículas para o detector.
- **Otimização de Mensagens:**
  - As mensagens entre atores devem ser concisas e conter apenas os dados necessários. Por exemplo, uma mensagem de "atualizar posição" de uma partícula para o detector de colisões pode conter apenas o PID da partícula e sua nova posição, em vez de todo o seu estado.
  - A natureza imutável das mensagens no Modelo de Ator [2] garante que não há efeitos colaterais inesperados quando as mensagens são passadas entre atores.
- **Gerenciamento de Estado do GenServer:**
  - O estado interno de cada GenServer é atualizado de forma sequencial, eliminando a necessidade de bloqueios explícitos para o estado interno do ator.[5]
  - Para operações que exigem atomicidade em um recurso compartilhado (como um contador global de colisões), :ets.update_counter/3 pode ser usado para garantir atualizações atômicas.[19]

A combinação do encapsulamento de estado do GenServer com as capacidades de armazenamento em memória de alto desempenho do ETS oferece uma estratégia robusta para gerenciar e otimizar os dados na simulação de colisões.

## 5. Otimização da Detecção de Colisões com Particionamento Espacial

A detecção de colisões é frequentemente o gargalo de desempenho em simulações com um grande número de objetos. Técnicas de particionamento espacial são essenciais para otimizar esse processo, reduzindo o número de pares de objetos que precisam ser verificados para uma possível colisão.

### 5.1. Hashing Espacial

Hashing Espacial é uma técnica de subdivisão espacial que mapeia objetos de uma região geométrica 3D em um número predefinido de "voxels" (células), o que ajuda a alcançar um desempenho próximo ao linear na detecção de colisões.[24] Esta técnica é particularmente eficaz para simulações com objetos deformáveis e garante um custo de atualização O(N) sem restrições de topologia.[24]

Os desafios do *hashing* espacial tradicional incluem:

- **Dispersão Desequilibrada:** Quando o espaço de mapeamento é reduzido, como quando objetos são comprimidos, a maioria dos triângulos pode ser compactamente mapeada em uma pequena porção de voxels, resultando em uma distribuição desigual.[24]
- **Colisões de Hash:** Funções de *hash* imperfeitas podem causar colisões, onde células não sobrepostas são mapeadas no mesmo voxel, levando a computações adicionais.[24]

Para mitigar esses problemas, uma técnica de "hashing de grupo" foi proposta. Esta abordagem visa reduzir a irregularidade do número de triângulos em voxels, melhorando o desempenho.[24] O método de

*hashing* de grupo equilibra a dispersão de células e o custo da subdivisão, podendo ser integrado a técnicas existentes de *hashing* espacial.[24]

O processo de *hashing* de grupo envolve três etapas [24]:

1. Cada triângulo é dividido pelo tamanho da célula, e seu índice de célula é encontrado usando uma função de *hash* espacial.
2. Um conjunto de possíveis índices de voxels para o triângulo é calculado. Se um desses voxels selecionados estiver vazio ou já contiver a célula, o triângulo é adicionado a esse voxel.
3. Se o triângulo ainda não foi adicionado, o número de triângulos em três voxels é comparado, e o triângulo é adicionado ao voxel com o menor número de triângulos, mapeando a célula para esse voxel.

Em uma arquitetura orientada a atores, o CollisionDetector.GenServer poderia ser o ator responsável por manter a estrutura de *hashing* espacial. Cada vez que recebe atualizações de posição das partículas, ele recalcula a célula e o voxel correspondente para cada partícula e atualiza a estrutura de dados interna. A busca por pares potenciais de colisão seria então restrita às partículas dentro do mesmo voxel ou voxels adjacentes, reduzindo drasticamente o número de verificações necessárias.

**5.2. Quadtrees**

Uma Quadtree é uma estrutura de dados em árvore na qual cada nó interno possui exatamente quatro filhos, sendo a análoga bidimensional de uma Octree.[25] Ela é mais frequentemente usada para particionar um espaço bidimensional, subdividindo-o recursivamente em quatro quadrantes ou regiões.[25] Os dados associados a uma célula folha variam conforme a aplicação, mas a célula folha representa uma "unidade de informação espacial interessante".[26]

Características comuns das Quadtrees incluem [26]:

- Decompõem o espaço em células adaptáveis.
- Cada célula (ou *bucket*) tem uma capacidade máxima. Quando a capacidade máxima é atingida, o *bucket* se divide.
- O diretório da árvore segue a decomposição espacial da Quadtree.

Existem diferentes formas de Quadtrees, como a *region quadtree* (que representa uma partição de espaço dividindo regiões em quatro quadrantes iguais) e a *point quadtree* (uma adaptação de uma árvore binária para representar dados de pontos bidimensionais, onde o centro de uma subdivisão está sempre em um ponto).[26]

Para a simulação de colisões, uma *region quadtree* seria mais apropriada. O CollisionDetector.GenServer manteria a Quadtree, onde cada nó folha representaria uma região espacial. As partículas seriam inseridas na Quadtree com base em suas posições. Quando uma célula folha atinge sua capacidade máxima (ou contém um número excessivo de partículas), ela se subdivide em quatro novos quadrantes. Isso permite uma adaptação dinâmica à densidade das partículas na simulação.

O processo de detecção de colisões usando uma Quadtree envolveria:

1. **Atualização da Quadtree:** Periodicamente, o CollisionDetector receberia as posições atualizadas das partículas e as inseriria (ou moveria) na Quadtree.
2. **Busca de Vizinhos:** Para cada partícula, o CollisionDetector buscaria outras partículas dentro da mesma célula folha da Quadtree e em células adjacentes.
3. **Testes de Colisão:** Apenas os pares de partículas encontrados nessas regiões seriam submetidos a testes de colisão precisos.

A Quadtree, assim como o Hashing Espacial, reduz significativamente o número de verificações de colisão, transformando um problema O(N^2) em algo mais próximo de O(N log N) ou O(N), dependendo da distribuição dos objetos. A escolha entre Hashing Espacial e Quadtree dependerá das características específicas da simulação, como a uniformidade da distribuição dos objetos e a natureza 2D/3D do ambiente. Para uma simulação 3D, uma Octree (a análoga 3D da Quadtree) seria a estrutura preferida.

## 6. Implementação e Considerações de Escalabilidade

A implementação da arquitetura orientada a atores em Elixir para uma simulação de colisões elásticas envolve a orquestração cuidadosa dos componentes OTP e a consideração de como o sistema escalará.

### 6.1. Estrutura da Aplicação Elixir

Uma aplicação Elixir para esta simulação seria estruturada como uma aplicação OTP, com uma árvore de supervisão bem definida. O mix new chat --sup é um comando inicial para gerar uma aplicação com um supervisor raiz.[11]

A estrutura da árvore de supervisão poderia ser:

- **Supervisor Raiz (SimulationApp.Application):** O supervisor principal da aplicação, definido no arquivo mix.exs com a chave mod: {SimulationApp.Application,}.[27]
    - **SimulationManager.GenServer:** O ator principal que orquestra a simulação.
    - **CollisionDetector.GenServer:** O ator responsável pela detecção de colisões, possivelmente supervisionado diretamente ou por um supervisor dedicado.
    - **CollisionResolver.GenServer:** O ator responsável pela resolução de colisões.
    - **Registry para Partículas:** Um Registry global (MyApp.Registry) para registrar e buscar PIDs de partículas por um identificador único, como um ID de partícula.[17]
    - **ParticleSupervisor (DynamicSupervisor):** Um supervisor dinâmico que gerencia a criação e o ciclo de vida dos Particle.GenServers. Isso permite adicionar ou remover partículas em tempo de execução de forma flexível.[15]
        - Particle.GenServer (múltiplas instâncias): Cada instância representa uma partícula individual na simulação.

Essa estrutura hierárquica garante que, se um Particle.GenServer individual falhar, ele será reiniciado pelo ParticleSupervisor sem afetar o restante da simulação.[8]

### 6.2. Otimização de Desempenho com Nx

Para os cálculos numéricos intensivos envolvidos na simulação de colisões (como atualização de posições, cálculo de distâncias, e resolução de colisões), o Numerical Elixir (Nx) oferece uma solução poderosa.[28] O Nx é uma biblioteca de tensores multidimensionais com compilação multiestágio para CPU/GPU, semelhante ao Numpy no ecossistema Python.[28]

A integração do Nx pode otimizar significativamente o desempenho:

- **Cálculos Vetoriais:** Em vez de realizar cálculos de posição e velocidade em cada Particle.GenServer individualmente usando operações escalares, o Nx permite que essas operações sejam vetorizadas. Um CollisionDetector ou SimulationManager poderia coletar os estados de um lote de partículas, convertê-los em tensores Nx, realizar os cálculos de movimento e colisão em massa (potencialmente em GPU) e então distribuir os resultados de volta para os Particle.GenServers.[28]
- **Compilação para CPU/GPU:** O Nx compila gráficos de computação para execução eficiente em CPUs ou GPUs, minimizando a comunicação de ida e volta entre o Elixir e o núcleo numérico.[28] Isso é crucial para simulações em tempo real onde a latência de cálculo é um fator crítico.
- **Nx.Serving para *Batching* e Balanceamento de Carga:** O Nx.Serving permite o *batching* de requisições de computação numérica e o balanceamento de carga em um *cluster* de máquinas.[28] Isso significa que o CollisionResolver poderia enviar lotes de colisões para um Nx.Serving para processamento paralelo e eficiente, aproveitando múltiplos núcleos ou nós distribuídos.

A imutabilidade, um pilar do Elixir, é uma ferramenta indispensável para construir esses gráficos de computação no Nx, tanto em termos de implementação quanto de raciocínio, um princípio também promovido pela biblioteca JAX do Google.[28]

### 6.3. Escalabilidade e Distribuição

A escalabilidade da simulação pode ser alcançada através da distribuição dos atores em múltiplos nós Elixir, aproveitando as capacidades de *clustering* da BEAM.[11]

- **Distribuição de Atores:**
  - **Particle.GenServers:** Podem ser distribuídos entre os nós do *cluster*. O Registry global facilitaria a localização de qualquer partícula, independentemente do nó em que ela esteja sendo executada.[17]
  - **CollisionDetector e CollisionResolver:** Para grandes simulações, pode ser benéfico ter múltiplas instâncias desses atores, cada uma responsável por uma região espacial específica ou por um subconjunto de colisões. Isso alavancaria ainda mais o particionamento espacial.
- **Phoenix.PubSub:** Para comunicação de *broadcast* (por exemplo, o SimulationManager enviando mensagens de "passo de tempo" ou o CollisionDetector publicando eventos de "colisão potencial"), o Phoenix.PubSub é uma escolha excelente.[9] Ele desacopla produtores e consumidores de mensagens e suporta distribuição entre nós.[1]
- **Monitoramento e Observabilidade:** Em um sistema distribuído, a depuração pode ser complexa.[7] Ferramentas de monitoramento e observabilidade, como o *observer* do Erlang, Telemetry (usado por bibliotecas como Finch [30]), e logging detalhado, são cruciais para entender o comportamento do sistema e diagnosticar problemas em tempo real.

A combinação da arquitetura de atores, das ferramentas OTP, do poder numérico do Nx e das capacidades de distribuição do Elixir/BEAM oferece uma base sólida para construir uma simulação de colisões elásticas altamente escalável, tolerante a falhas e performática.

## 7. Conclusões

A análise detalhada da Arquitetura Orientada a Atores em Elixir para simulações de colisões elásticas concorrentes revela que essa abordagem é não apenas viável, mas intrinsecamente adequada para os desafios inerentes a sistemas complexos e dinâmicos. A força motriz por trás dessa adequação reside na filosofia de "não compartilhamento de estado mutável" e na comunicação assíncrona por meio de mensagens, princípios que simplificam fundamentalmente a concorrência e promovem a resiliência.

A capacidade do Elixir, alavancando a Máquina Virtual Erlang (BEAM), de materializar esses princípios por meio de processos leves e eficientes é um diferencial crucial. A BEAM não é meramente um ambiente de execução; é uma plataforma que foi projetada especificamente para operar sistemas altamente concorrentes e distribuídos com notável tolerância a falhas. Isso permite que a simulação de colisões elásticas não apenas gerencie um grande número de partículas de forma eficiente, mas também se recupere graciosamente de falhas localizadas, um atributo vital para a estabilidade de simulações de longa duração.

A modelagem de entidades da simulação como atores (GenServers) — desde partículas individuais até componentes de detecção e resolução de colisões — oferece uma estrutura modular e escalável. A utilização de ferramentas OTP, como as Árvores de Supervisão, garante que a simulação possa se auto-curar em caso de falhas de atores, mantendo a integridade e o progresso da simulação. O Registry e o Phoenix.PubSub fornecem mecanismos robustos para a descoberta e comunicação transparente entre atores, independentemente de sua localização física no *cluster*.

Os desafios de desempenho na detecção de colisões, que tipicamente escalam quadraticamente com o número de objetos, são efetivamente mitigados pela integração de técnicas de particionamento espacial, como Hashing Espacial ou Quadtrees. Essas estruturas, gerenciadas por atores dedicados, reduzem drasticamente o número de pares de objetos que precisam ser verificados, transformando um problema computacionalmente proibitivo em um gerenciável. Além disso, a emergência do Numerical Elixir (Nx) oferece uma otimização sem precedentes para os cálculos numéricos da simulação, permitindo a vetorização de operações e a execução acelerada em CPUs e GPUs. Isso significa que as complexas equações de colisão elástica podem ser resolvidas com alta performance, permitindo simulações mais realistas e em maior escala.

Em suma, a arquitetura orientada a atores em Elixir para simulações de colisões elásticas não é apenas uma escolha técnica, mas uma estratégia arquitetural que alinha as necessidades de alta concorrência e tolerância a falhas de simulações complexas com um ecossistema de desenvolvimento maduro e performático. A capacidade de escalar horizontalmente, a resiliência inerente e a otimização de cálculos numéricos posicionam o Elixir como uma ferramenta poderosa para o futuro das simulações físicas concorrentes.

**Referências citadas**


1. Actor model - Wikipedia, acessado em junho 11, 2025, https://en.wikipedia.org/wiki/Actor_model
2. Actor Model Explained - Finematics, acessado em junho 11, 2025, https://finematics.com/actor-model-explained/
3. What is the actor model and when should I use it? | Stately - Stately.ai, acessado em junho 11, 2025, https://stately.ai/blog/what-is-the-actor-model
4. Introduction to Actor Model - Ada Beat, acessado em junho 11, 2025, https://adabeat.com/fp/introduction-to-actor-model/
5. Understanding the Actor Model - MentorCruise, acessado em junho 11, 2025, https://mentorcruise.com/blog/understanding-the-actor-model/
6. Actor Model For Object Oriented Programmers | HackerNoon, acessado em junho 11, 2025, https://hackernoon.com/actor-model-for-object-oriented-programmers
7. Actor Model in Distributed Systems | GeeksforGeeks, acessado em junho 11, 2025, https://www.geeksforgeeks.org/actor-model-in-distributed-systems/
8. Exploring Elixir's OTP Supervision Trees - CloudDevs, acessado em junho 11, 2025, https://clouddevs.com/elixir/otp-supervision-trees/
9. www.pompecki.com, acessado em junho 11, 2025, https://www.pompecki.com/post/phoenix-pubsub/#:~:text=Meet%20PubSub&text=The%20producer%20of%20the%20message,between%20Elixir%20processes%20with%20Phoenix.
10. Phoenix PubSub - pompecki.com, acessado em junho 11, 2025, https://www.pompecki.com/post/phoenix-pubsub/
11. OTP Distribution - Elixir School, acessado em junho 11, 2025, https://elixirschool.com/en/lessons/advanced/otp_distribution
12. Exploring Concurrency in Elixir with The Actor Model, acessado em junho 11, 2025, https://elixirmerge.com/p/exploring-concurrency-in-elixir-with-the-actor-model
13. Overview — Erlang System Documentation v28.0, acessado em junho 11, 2025, https://www.erlang.org/doc/system/design_principles.html
14. Mastering GenServer for Enhanced Elixir Applications - Curiosum, acessado em junho 11, 2025, https://curiosum.com/blog/what-is-elixir-genserver
15. Elixir Full Course: 40 - Supervisors - YouTube, acessado em junho 11, 2025, https://www.youtube.com/watch?v=33ORRGDb1Jo
16. Quick guide to creating Elixir supervision trees from scratch - GitHub Gist, acessado em junho 11, 2025, https://gist.github.com/gkaemmer/12a536f7c859c576200e974235e2f923
17. Exploring Elixir's Registry: Dynamic Process Groups - Cloud Devs, acessado em junho 11, 2025, https://clouddevs.com/elixir/registry/
18. Understanding {:via, Registry, {...}} vs Registry ... - DEV Community, acessado em junho 11, 2025, https://dev.to/marknefedov/understanding-via-registry-vs-registryregister3-in-elixir-56go
19. Erlang Term Storage (ETS) · Elixir School, acessado em junho 11, 2025, https://elixirschool.com/en/lessons/storage/ets
20. Erlang Term Storage · Elixir Phoenix Web Stack, acessado em junho 11, 2025, https://essenceofchaos.gitbooks.io/elixir-phoenix-web-stack/content/section-ii/erlang-term-storage.html
21. Clustering Your Application · Fly Docs - Fly.io, acessado em junho 11, 2025, https://fly.io/docs/elixir/the-basics/clustering/
22. Using :dns_cluster with docker-compose locally (it can be done) - Elixir Forum, acessado em junho 11, 2025, https://elixirforum.com/t/using-dns-cluster-with-docker-compose-locally-it-can-be-done/61336
23. Simulation Manual: Elastic Collision in One Dimension - Physics Zone, acessado em junho 11, 2025, https://physics-zone.com/simulation-manual-elastic-collision-in-one-dimension-en/
24. balanced spatial subdivision method for continuous collision ..., acessado em junho 11, 2025, https://graphics.ucdenver.edu/publications/Balanced-Spatial-Subdivision-Method-for-Continuous-Collision-Detection.pdf
25. en.wikipedia.org, acessado em junho 11, 2025, https://en.wikipedia.org/wiki/Quadtree#:~:text=A%20quadtree%20is%20a%20tree,into%20four%20quadrants%20or%20regions.
26. Quadtree - Wikipedia, acessado em junho 11, 2025, https://en.wikipedia.org/wiki/Quadtree
27. How OTP Applications are structured - AppSignal Blog, acessado em junho 11, 2025, https://blog.appsignal.com/2018/09/18/elixir-alchemy-how-otp-applications-are-structured.html
28. Numerical Elixir (Nx) · GitHub, acessado em junho 11, 2025, https://github.com/elixir-nx
29. Machine Learning in Elixir, acessado em junho 11, 2025, https://unidel.edu.ng/focelibrary/books/Machine%20Learning%20in%20Elixir%20(Sean%20Moriarity)%20(Z-Library).pdf
30. sneako/finch: Elixir HTTP client, focused on performance - GitHub, acessado em junho 11, 2025, https://github.com/sneako/finch