ERLDA: Explorando Concorrência e Resiliência com SEDA em Erlang

Fernando Areias

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Avenida Sete de Setembro, 3165 – Rebouças – 80.230-901 – Curitiba – PR – Brazil

Abstract. *The scalability and efficiency of web servers are essential to meet the growing demand for digital services. Solutions that combine high connection density with low operational costs are increasingly necessary. In this context, web servers developed with Erlang, based on the SEDA (Staged Event-Driven Architecture) framework, present a promising alternative. Erlang's BEAM VM, designed to support millions of simultaneous processes with efficient resource usage, stands out as an efficient solution in scenarios that prioritize vertical scalability and operational simplicity. Traditional web servers, based on synchronous TCP connections, face significant limitations in high-demand scenarios, especially when managing millions of simultaneous connections. The scalability of these solutions often depends on distributed clusters, which increases both cost and complexity. The SEDA architecture, combined with Erlang, offers an asynchronous and concurrent approach that efficiently manages lightweight processes, reducing overhead and optimizing resource usage. This highlights the need to explore solutions that overcome the limitations of traditional models, providing greater efficiency and performance in high-demand environments. This paper aims to develop a scalable and resilient web server, using the SEDA architecture with Erlang and its OTP framework. The proposal is to create an efficient and cost-effective solution capable of meeting the growing demands for scalability, offering a robust alternative compared to traditional approaches. Experiments were conducted in a controlled environment to evaluate the scalability and performance of the proposed architecture. During the tests, metrics such as response time, simultaneous connection rate, and resource usage were collected to validate the server's capacity under different load conditions. No results were obtained. The proposal was to develop a scalable and resilient web server, using the SEDA architecture and Erlang with the OTP framework. The solution aims to manage large volumes of simultaneous connections with high availability and reduced costs, offering an efficient alternative compared to traditional approaches, addressing the growing demand for high-performance digital services.*
Note: the code for the thread-based Java server is available at https://github.com/fernandoareias/web-server, and the code for the event-based Erlang server is available at https://github.com/fernandoareias/web-server-erl.

Resumo. *A escalabilidade e eficiência dos servidores web são fundamentais para atender à crescente demanda por serviços digitais. Soluções que combinam alta densidade de conexões com baixo custo operacional são cada*

vez mais necessárias. Nesse contexto, servidores web desenvolvidos com Erlang, baseados na arquitetura SEDA (Staged Event-Driven Architecture), se apresentam como uma alternativa promissora. A BEAM VM do Erlang, projetada para suportar milhões de processos simultâneos com uso eficiente de recursos, destaca-se como uma solução eficiente em cenários que priorizam escalabilidade vertical e simplicidade operacional. Servidores web tradicionais, baseados em conexões TCP síncronas, têm limitações significativas em cenários de alta demanda, especialmente na gestão de milhões de conexões simultâneas. A escalabilidade dessas soluções frequentemente depende de clusters distribuídos, o que aumenta os custos e a complexidade. A arquitetura SEDA, combinada com Erlang, oferece uma abordagem assíncrona e concorrente que gerencia processos leves de maneira eficiente, reduzindo o overhead e otimizando o uso de recursos. Isso destaca a necessidade de explorar soluções que superem as limitações dos modelos tradicionais, proporcionando maior eficiência e desempenho em ambientes de alta demanda. Este artigo tem como objetivo desenvolver um servidor web escalável e resiliente, utilizando a arquitetura SEDA com Erlang e seu framework OTP. A proposta é criar uma solução eficiente e econômica, capaz de atender a crescentes demandas por escalabilidade, oferecendo uma alternativa robusta em comparação com abordagens tradicionais. Foram realizados experimentos em ambiente controlado para avaliar a escalabilidade e o desempenho da arquitetura proposta. Durante os testes, métricas como tempo de resposta, taxa de conexões simultâneas e uso de recursos foram coletadas para validar a capacidade do servidor em diferentes condições de carga. Não tem resultado. Foi proposto o desenvolvimento de um servidor web escalável e resiliente, utilizando a arquitetura SEDA e Erlang com o framework OTP. A solução visa gerenciar grandes volumes de conexões simultâneas com alta disponibilidade e custos reduzidos, oferecendo uma alternativa eficiente em comparação com abordagens tradicionais, atendendo às crescentes demandas por serviços digitais de alto desempenho.

Obs: código do servidor java baseado em threads está disponível em https://github.com/fernandoareias/web-server e código do servidor erlang baseado em eventos está disponível em https://github.com/fernandoareias/web-server-erl

1. Introdução

A Internet tornou-se indispensável para a vida moderna, cenário que se intensificou com a pandemia de 2020. Durante esse período, a demanda por serviços digitais cresceu significativamente, acompanhada pelo aumento do número de usuários de internet no Brasil. Segundo dados do *Cetic.br* (Cetic.br 2021), o percentual de usuários de internet na população com 10 anos ou mais passou de 74% em 2019 para 81% em 2020, representando um crescimento de 7 pontos percentuais. Grande parte desses serviços é fornecido por servidores web, que desempenham um papel crucial na infraestrutura da Internet.

1.1. Servidores Web

Servidores web são softwares projetados para processar solicitações de usuários ou aplicações, disponibilizando páginas web e serviços de forma eficiente por meio da rede. Eles desempenham um papel crítico na Internet moderna, servindo como a base para a entrega de conteúdo e funcionalidades que sustentam desde websites simples até aplicações de grande escala.

Como apresentado por (Drolia and Wang nd), os servidores web modernos enfrentam três desafios fundamentais para alcançar desempenho em escala. Primeiramente, a alta concorrência, que exige que os servidores sejam capazes de lidar simultaneamente com milhares ou até milhões de conexões de usuários sem comprometer a estabilidade ou o desempenho. Em segundo lugar, o alto throughput, que demanda que os servidores processem grandes volumes de solicitações de clientes de forma eficiente e responsiva, garantindo uma experiência satisfatória para os usuários finais. Por fim, os servidores precisam estar preparados para lidar com grandes variações na carga, como os picos inesperados de tráfego decorrentes do aumento súbito de popularidade de um site — fenômeno conhecido como *Slashdot Effect* (Adler 1999). Esses desafios destacam a complexidade e a importância de arquiteturas robustas e escaláveis para atender às demandas da web moderna.

Nos últimos anos, diversas propostas têm sido apresentadas para superar esses desafios, sendo duas técnicas particularmente proeminentes: a programação concorrente baseada em threads e a programação concorrente baseada em eventos. Existe um intenso debate sobre os prós e contras de ambas as abordagens.

1.2. Threads

A capacidade de processar informações de forma concorrente é fundamental para permitir que servidores web atendam a múltiplos usuários simultaneamente em uma única máquina. Essa concorrência é essencial no contexto de sistemas modernos, onde a demanda por alta disponibilidade e escalabilidade é cada vez maior. Uma das abordagens clássicas para alcançar concorrência é o uso de threads, que possibilita a execução paralela de diferentes tarefas em um único processo.

No modelo baseado em threads, cada requisição de um cliente resulta na criação de uma nova thread para processar a conexão. Esse modelo é atrativo pela simplicidade de implementação e pela capacidade de aproveitar o suporte nativo das linguagens modernas para threads. O exemplo a seguir ilustra uma implementação básica de servidor web em Java, onde cada conexão recebida é delegada a uma thread separada para processamento

```
1  package com.fernando;
2  import java.io.IOException;
3  import java.net.*;
4
5  public class Main {
6      public static void main(String[] args) throws IOException {
7          // Porta que o servidor ir  ouvir para receber conex es
8          final int PORT = 8080;
9
10         System.out.println("Iniciando o servidor web...");
```

```
11
12          // Inicia o servidor socket na porta definida (8080)
13          try (var socketServ = new ServerSocket(PORT)) {
14              Socket socketCli;
15
16              // Mant m o servidor ativo para aceitar conex es
                    continuamente
17              while (true) {
18
19                  System.out.println("Servidor ativo, aguardando
                        conex es...");
20
21                  // Aguarda e aceita uma conex o de um cliente
22                  socketCli = socketServ.accept();
23
24                  // Cria uma inst ncia da requisi  o HTTP
                        associada ao cliente conectado
25                  HttpRequest requisicao = new HttpRequest(socketCli)
                        ;
26
27                  // Cria uma nova thread para processar a
                        requisi  o de forma concorrente
28                  Thread thread = new Thread(requisicao);
29
30                  // Inicia a thread para realizar o processamento da
                         requisi  o
31                  thread.start();
32
33                  // O servidor volta ao in cio do loop para
                        aguardar outras conex es
34              }
35          }
36      }
37  }
```

Listing 1. Exemplo de servidor web baseado em threads.

Embora essa abordagem seja funcional, apresenta desafios significativos quando se trata de sistemas de larga escala. Um dos principais problemas é a sobrecarga de recursos associada à criação e gerenciamento de threads. Cada thread consome memória e impõe custos ao sistema operacional para alternar entre elas, o que pode levar à degradação do desempenho conforme o número de conexões simultâneas aumenta.

Além disso, o uso de threads exige que os desenvolvedores implementem mecanismos de sincronização para evitar condições de corrida e outros problemas de concorrência, o que aumenta a complexidade do código e o risco de bugs. Em sistemas com alta carga, essa abordagem pode se tornar insustentável, exigindo arquiteturas alternativas que sejam mais eficientes e escaláveis.

1.3. Eventos

A programação concorrente baseada em eventos é uma abordagem que utiliza uma única thread para processar uma fila de eventos, onde cada requisição de cliente é

enfileirada e tratada sequencialmente como uma série de eventos. Essa estratégia elimina a necessidade de criar uma nova thread para cada solicitação, reduzindo significativamente o consumo de recursos e simplificando a gerência de threads.

Essa abordagem se destaca por seu baixo custo de gerenciamento de estado e pela capacidade de lidar eficientemente com uma grande quantidade de conexões simultâneas em sistemas com alta densidade de eventos, como servidores web. No entanto, ao longo dos anos, diversas limitações do modelo baseado em eventos foram identificadas. Drolia e Wang (Drolia and Wang nd) apontam que uma das principais dificuldades está na complexidade associada ao *stack ripping*, ou *callback hell*, um padrão em que o código é fragmentado em múltiplas funções de retorno (callbacks), muitas vezes gerando longas cadeias de dependências e dificultando a manutenção.

Outra desvantagem significativa é a necessidade de gerenciamento manual do estado entre pares de chamadas e retornos. Embora esse controle possa ser útil para otimizações específicas, ele representa um fardo adicional para os desenvolvedores, que precisam lidar com a persistência e a recuperação do estado de forma explícita. Por fim, sistemas baseados em eventos apresentam limitações inerentes em tirar proveito do paralelismo real da CPU, especialmente em arquiteturas multi-core, onde a utilização de múltiplas threads pode ser mais eficiente.

No código abaixo, apresentamos um exemplo de servidor web baseado em eventos, implementado em Erlang. Aqui, cada conexão é gerenciada como um evento recebido por um processo responsável por enfileirar e despachar as mensagens para filas específicas:

```
1
2  init([]) ->
3      io:format("Web server request processor listening...~n"),
4
5      case gen_tcp:listen(8091, [{active, true}, binary]) of
6          {ok, ListenSocket} ->
7              io:format("Web server accepts messages now!~n"),
8              loop(ListenSocket);
9          {error, eaddrinuse} ->
10             io:format("Error port already in use~n");
11         _ ->
12             io:format("Error when start web server~n"),
13             error
14     end.
15
16 loop(ListenSocket) ->
17     {ok, AcceptSocket} = gen_tcp:accept(ListenSocket),
18     io:format("Connection accepted: ~p at ~p~n", [AcceptSocket,
           calendar:local_time()]),
19
20     receive
21         {tcp, AcceptSocket, Data} ->
22             io:format("Received message: ~s~n", [Data]),
23             io:format("Send data to event queue~n~n"),
24             gen_server:cast(web_server_request_queue, {Data,
                   AcceptSocket}),
25             loop(ListenSocket);
26         {tcp_closed, AcceptSocket} ->
```

```
27              io:format("Connection closed: ~p~n", [AcceptSocket]),
28              loop(ListenSocket);
29          {tcp_error, AcceptSocket, Reason} ->
30              io:format("Connection error: ~p~n", [Reason]),
31              loop(ListenSocket)
32      end.
33  .
```

Listing 2. Exemplo de servidor web baseado em eventos.

Nesse exemplo, o processo principal escuta a porta 8091, aceitando conexões de clientes e manipulando os dados recebidos de forma assíncrona. A fila de eventos é gerenciada de forma eficiente por um processo dedicado, reduzindo a complexidade associada ao gerenciamento de múltiplas threads. Cada evento é despachado para uma fila específica por meio do **gen server**, que encapsula o estado necessário para processar a mensagem.

1.4. SEDA - Staged Event Driven-Architecture

A Staged Event-Driven Architecture (SEDA) é um modelo arquitetural projetado para melhorar a escalabilidade, modularidade e robustez de sistemas computacionais altamente concorrentes. Conforme descrito por Welsh et al. (Welsh et al. 2001), SEDA organiza a aplicação em uma série de estágios, cada um correspondendo a uma unidade lógica de processamento. Esses estágios se comunicam entre si através de filas assíncronas, permitindo que o sistema desacople as operações de entrada e saída de suas respectivas tarefas de processamento.

Um dos principais benefícios do SEDA é sua capacidade de controlar a sobrecarga e o consumo de recursos do sistema. Welsh et al. (Welsh et al. 2001) destacam que cada estágio pode ser configurado para operar com limites de carga específicos, otimizando o uso de threads e prevenindo gargalos que poderiam comprometer o desempenho do sistema. Essa abordagem permite que o sistema degrade graciosamente sob condições de alta carga, mantendo a responsividade.

Além disso, SEDA facilita a modularidade e a facilidade de manutenção. Cada estágio encapsula uma funcionalidade específica e pode ser desenvolvido e testado de forma independente. Isso não apenas promove a reutilização de componentes, mas também simplifica a identificação e correção de falhas no sistema, como apontado por Welsh et al. (Welsh et al. 2001).

Outro aspecto relevante do modelo SEDA é sua abordagem orientada a eventos. Os eventos são tratados de maneira assíncrona em cada estágio, permitindo um processamento eficiente e escalável. No entanto, Welsh et al. (Welsh et al. 2001) também discutem algumas limitações, como a complexidade adicional na coordenação entre estágios e o desafio de lidar com tarefas que exigem processamento síncrono ou estado compartilhado.

O presente trabalho busca superar as limitações do modelo SEDA, bem como outras restrições inerentes ao modelo baseado em eventos, por meio do uso de programação funcional. Para isso, optou-se pela linguagem Erlang, reconhecida por seu design robusto e altamente adequado para sistemas concorrentes e distribuídos.

Além disso, foi desenvolvido um servidor web dedicado à coleta de métricas da implementação, permitindo uma análise detalhada do desempenho e da escalabilidade do sistema proposto.

1.5. Erlang

Erlang é uma linguagem funcional projetada para criar sistemas concorrentes, distribuídos e de alta disponibilidade. Desenvolvida pela Ericsson na década de 1980, Erlang é executada na BEAM (*Bogdan/Björn's Erlang Abstract Machine*), uma máquina virtual otimizada para lidar com processos leves e escalabilidade horizontal. A BEAM fornece suporte nativo à concorrência, com milhões de processos isolados que podem se comunicar via troca de mensagens, além de mecanismos de gerenciamento de falhas, fundamentais para sistemas críticos.

1.6. BEAM

O BEAM é a máquina virtual usada para executar código Erlang e Elixir, sendo projetada com foco em concorrência massiva, tolerância a falhas e desempenho distribuído. Inspirado pela filosofia de construção de sistemas resilientes, Joe Armstrong destacou que "o modelo de processos isolados e comunicação por troca de mensagens é essencial para alcançar sistemas robustos e distribuídos" (Armstrong 2003). O BEAM implementa essa arquitetura, permitindo que milhares de processos leves, independentes e supervisionados coexistam, cada um com sua própria pilha e heap, otimizando o uso de recursos do sistema e a escalabilidade em ambientes de múltiplos nós.

1.7. Supervisores

Um dos elementos centrais da arquitetura de Erlang é o modelo de supervisores. Supervisores são processos especializados em monitorar outros processos, denominados "filhos", com o objetivo de garantir que falhas não comprometam o funcionamento do sistema. Quando um processo filho apresenta um erro ou falha, o supervisor intervém automaticamente, aplicando estratégias de recuperação predefinidas. Entre as estratégias disponíveis estão o reinício individual, que reinicia apenas o processo específico que falhou, e o reinício em cascata, que reinicia um conjunto de processos interdependentes. Esse modelo, incorporado por design em Erlang, é uma das principais razões pelas quais a linguagem é amplamente utilizada em sistemas críticos que exigem alta disponibilidade e resiliência.

No trabalho de Welsh et al. (Welsh et al. 2001), os autores destacam a dificuldade e a complexidade de implementar mecanismos semelhantes para monitorar e controlar o estado das filas de eventos em Java. A ausência de suporte nativo para supervisores em linguagens tradicionais exige que os desenvolvedores criem soluções personalizadas, resultando em um aumento na complexidade do código e no risco de falhas. Em contrapartida, o modelo de supervisores de Erlang oferece essa funcionalidade de forma integrada, simplificando a gestão de falhas e tornando os sistemas mais robustos por padrão.

Neste trabalho, estendemos a funcionalidade dos supervisores de Erlang para implementar um supervisor especializado no monitoramento do processo responsável

pela fila de eventos. Esse supervisor será encarregado de observar o comportamento da fila, garantir que eventos sejam processados corretamente e intervir em caso de falhas, aplicando estratégias de recuperação adequadas. Essa abordagem não apenas melhora a robustez da implementação, mas também demonstra como os conceitos fundamentais de Erlang podem ser adaptados e ampliados para atender a requisitos específicos de sistemas modernos e distribuídos.

2. Estado da Arte

A Arquitetura Dirigida por Eventos em Estágios (SEDA) foi introduzida por Welsh, Culler e Brewer em 2001 como uma abordagem para projetar aplicações servidoras altamente concorrentes. SEDA organiza aplicações como uma rede de estágios, cada um associado a uma fila de eventos, proporcionando isolamento entre componentes e facilitando a adaptação dinâmica a variações de carga (Welsh et al. 2001). Este modelo emprega controladores de recursos dinâmicos, como ajuste do tamanho de pools de threads e descarte adaptativo de carga, assegurando que o sistema permaneça funcional mesmo sob grandes flutuações de demanda. Tais características tornam a arquitetura robusta para lidar com cenários de alta concorrência e picos de demanda, promovendo escalabilidade e estabilidade.

No trabalho de Gordon (2010), intitulado *Stage Scheduling for CPU-intensive Servers*, uma nova abordagem para o agendamento de estágios foi proposta, voltada para servidores com alta demanda de CPU. Em vez de utilizar pools de threads por estágio, como na proposta de SEDA, a abordagem de Gordon sugere o uso de um thread único por núcleo físico, controlado em nível de usuário (Gordon 2010). Essa técnica evita a sobrecarga de threads e melhora a eficiência de uso de recursos, resultando em maior throughput e menor latência. Tais avanços destacam-se como uma alternativa eficiente para servidores modernos que necessitam de alta performance em hardware multicore.

Inspirados pela SEDA, Salmito et al. (2013) propuseram a LEDA (*Lua Estaged Event-Driven Architecture*), que busca maior flexibilidade e desacoplamento no design de aplicações (Salmito et al. 2013). A LEDA utiliza conectores assíncronos para interligar os estágios, permitindo que a lógica de cada estágio seja reutilizada em diferentes contextos. Além disso, introduz o conceito de clusters, que permitem particionar o grafo de estágios em unidades menores, mapeadas para processos distintos e distribuíveis entre hosts. Essas inovações promovem maior granularidade no escalonamento e flexibilidade no desenvolvimento, sendo especialmente relevantes em aplicações distribuídas e de alto desempenho.

Na sequência, Salmito, Moura e Rodriguez (2014) aprofundaram o modelo LEDA, propondo uma abordagem baseada em grafos para representar a estrutura das aplicações (Salmito et al. 2014). Os nós do grafo representam estágios independentes conectados por canais de comunicação assíncronos, chamados conectores. Este modelo facilita a exploração de diferentes configurações de execução sem alterar a lógica da aplicação, o que é particularmente útil para servidores web de alta escalabilidade. A possibilidade de distribuir clusters em diferentes hosts aumenta ainda mais o potencial de escalabilidade da abordagem.

Outro avanço relevante foi apresentado por Cruz (2015), que introduziu uma

interface de programação para controle de sobrecarga em arquiteturas baseadas em estágios (Cruz 2015). A LEDA, implementada em Lua, foi estendida para permitir políticas dinâmicas de controle de recursos, reduzindo a ociosidade das threads e otimizando o uso de recursos em diferentes cenários de execução. A aplicação desta arquitetura em servidores HTTP demonstrou melhorias significativas na utilização de recursos e gestão de estágios, destacando a flexibilidade e adaptabilidade do modelo.

Com base nesses avanços apresentados na literatura, este trabalho propõe a aplicação dos conceitos da arquitetura SEDA na linguagem Erlang. A escolha por Erlang justifica-se por suas características intrínsecas de concorrência, isolamento de processos e tolerância a falhas, que a tornam ideal para o desenvolvimento de servidores web de alta escalabilidade. A abordagem proposta inspira-se nos conceitos introduzidos pela LEDA, especialmente no uso de conectores assíncronos e no desacoplamento entre estágios, mas busca adaptar essas ideias ao contexto moderno de servidores distribuídos. Adicionalmente, o modelo de atores e a comunicação assíncrona nativa de Erlang são explorados para ampliar a resiliência e a eficiência do sistema proposto.

Table 1. Comparação entre Trabalhos Relacionados e Este Trabalho

Trabalho	**Facilidade para trabalhar com multi-threads**	**Facilidade para impedir compartilhamento da memória**	**Facilidade para implementar diferentes escalonadores**	**Fraco acoplamento entre estágios**	**Execução distribuída entre hosts**
SEDA: An Architecture for Well-Conditioned, Scalable Internet Services (Welsh et al., 2001)	Não	Não	Sim	Não	Não
Stage Scheduling for CPU-intensive Servers (Gordon, 2010)	Não	Não	Sim	Não	Não
A Flexible Approach to Staged Events (LEDA - Salmito et al., 2013)	Não	Não	Sim	Sim	Sim
A Stepwise Approach to Developing Staged Applications (Leda Estendida - Salmito et al., 2014)	Não	Sim (parcialmente)	Sim	Sim	Sim
Uma Interface de Programação para Controle de Sobrecarga em Arquiteturas Baseadas em Estágios (Cruz, 2015)	Não	Não	Sim	Sim	Não
Esse trabalho	Sim	Sim	Sim	Sim	Sim

References

[Adler 1999] Adler, B. (1999). The slashdot effect: An analysis of popularity surges. *Online Systems Journal*, 12(3):45–50. Accessed: 2024-12-18.

[Armstrong 2003] Armstrong, J. (2003). *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, KTH, Stockholm, Sweden.

[Cetic.br 2021] Cetic.br (2021). TIC Domicílios 2020: Pesquisa sobre o uso das tecnologias de informação e comunicação nos domicílios brasileiros. Acessado em: 19 dez. 2024.

[Cruz 2015] Cruz, F. (2015). *Uma Interface de Programação para Controle de Sobrecarga em Arquiteturas Baseadas em Estágios*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio).

[Drolia and Wang nd] Drolia, V. and Wang, C. A. C. S. (n.d.). Threads vs events for server architectures. Accessed: 2024-12-18.

[Gordon 2010] Gordon, M. E. (2010). Stage scheduling for cpu-intensive servers. *UCAM-CL-TR-781, Technical Report, University of Cambridge*.

[Salmito et al. 2013] Salmito, T., de Moura, A. L., and Rodriguez, N. (2013). A flexible approach to staged events. In *Proceedings of the 42nd International Conference on Parallel Processing*, pages 661–670. IEEE.

[Salmito et al. 2014] Salmito, T., de Moura, A. L., and Rodriguez, N. (2014). A stepwise approach to developing staged applications. *The Journal of Supercomputing*, 70(1):42–61.

[Welsh et al. 2001] Welsh, M., Culler, D., and Brewer, E. (2001). Seda: An architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243.