ERLDA: Explorando Concorrência e Resiliência com SEDA em Erlang

Fernando Areias, Adolfo Neto

Departamento Acadêmico de Informática (DAINF)
Universidade Tecnológica Federal do Paraná (UTFPR)
Av. Sete de Setembro, 3165 - Rebouças
CEP 80230-901 - Curitiba - PR - Brazil

fernandoareias@alunos.utfpr.edu.br, adolfo@utfpr.edu.br

Abstract. *Staged Event-Driven Architecture (SEDA) has been widely studied for developing highly concurrent systems, offering modularity and overload control through asynchronous queues and dynamic resource adjustment. However, its adaptation to functional languages with different concurrency models remains limited. In this context, this work proposes ERLDA, an implementation in the concurrent functional language Erlang, inspired by the Lua Staged Event-Driven Architecture (LEDA). LEDA is a Lua library for building parallel and non-linear pipelines based on SEDA concepts. Our solution leverages lightweight processes, asynchronous message passing, and native supervision mechanisms to build a modular and resilient architecture. Each stage encapsulates processing logic and includes a controller that automatically adjusts the number of workers. The ERLDA design shows promising potential for applications in scalable distributed systems.*

Resumo. *A Arquitetura baseada em Estágios (SEDA) tem sido amplamente estudada para o desenvolvimento de sistemas altamente concorrentes, oferecendo modularidade e controle de sobrecarga por meio de filas assíncronas e ajuste dinâmico de recursos. No entanto, sua adaptação a linguagens funcionais com modelos de concorrência distintos ainda é limitada. Nesse contexto, este trabalho propõe o ERLDA, uma implementação na linguagem funcional concorrente Erlang, inspirada na Lua Staged Event-Driven Architecture (LEDA). LEDA é uma biblioteca em Lua para construir pipelines paralelos e não lineares com base nos conceitos da SEDA. Nossa solução explora processos leves, comunicação assíncrona e mecanismos nativos de supervisão para construir uma arquitetura modular e resiliente. Cada estágio encapsula lógica de processamento e conta com um controlador que ajusta automaticamente o número de workers. O design do ERLDA sugere um potencial promissor para aplicações em sistemas distribuídos escaláveis.*

1. Introdução

A arquitetura de software é um dos pilares da Engenharia de Software, desempenhando um papel fundamental no desenvolvimento de sistemas robustos, escaláveis e eficientes. Neste contexto, a Arquitetura Baseada em Estágios (SEDA) se destaca por organizar sistemas em estágios conectados por filas assíncronas, oferecendo modularidade, controle

de sobrecarga e ajuste dinâmico de recursos [Welsh et al. 2001]. Apesar de sua ampla adoção em linguagens tradicionais, sua adaptação para linguagens funcionais com modelos de concorrência distintos ainda é pouco explorada.

Este artigo apresenta o ERLDA[1], uma implementação da SEDA na linguagem funcional Erlang, que combina os princípios de modularidade e escalabilidade da SEDA com as características únicas de Erlang, como processos leves, comunicação assíncrona e tolerância a falhas. Inspirado na biblioteca LEDA (*Lua Staged Event-Driven Architecture*) [Salmito et al. 2013], o ERLDA explora mecanismos nativos de supervisão e balanceamento dinâmico de recursos para construir uma arquitetura modular e resiliente. Acreditamos que esta abordagem não apenas enriquece o estado da arte em arquiteturas de software, mas também oferece um potencial promissor para sistemas distribuídos escaláveis.

2. Arquitetura Baseada em Estágios (SEDA)

A Arquitetura Baseada em Estágios (SEDA) foi introduzida como uma abordagem para o desenvolvimento de sistemas altamente concorrentes. Ela organiza aplicações em estágios interconectados por filas assíncronas, proporcionando isolamento entre componentes e permitindo o ajuste dinâmico de recursos [Welsh et al. 2001].

2.1. Estágios

Estágio é a unidade mais básica de processamento. Cada estágio é composto por um manipulador de eventos, responsável por processar lotes de eventos provenientes de uma fila associada ao estágio. O processamento é realizado com um *pool* de *threads* que gerencia a execução concorrente das tarefas. Cada estágio é controlado dinamicamente por um controlador, que ajusta automaticamente aspectos como agendamento e alocação de novas *threads* no *pool*. Essa gestão dinâmica permite que o sistema se adapte às condições de carga, otimizando recursos e garantindo eficiência. O *pool* de *threads* opera consumindo eventos da fila em formato de lote, entregando-os ao manipulador para processamento. Após o processamento, o manipulador envia os eventos resultantes para a fila do próximo estágio, continuando o ciclo de processamento [Welsh et al. 2001].

2.2. Fila de Eventos

A fila de eventos desempenha um papel crucial no armazenamento temporário dos eventos que serão processados pelo *pool* de *threads*. O design do SEDA oferece flexibilidade para implementar diferentes políticas de ordenação e processamento. Uma característica importante dessas filas é sua natureza finita, ou seja, cada fila possui um limite máximo de eventos que pode armazenar. Quando esse limite é atingido, o sistema precisa lidar com novos eventos de forma controlada. Para isso, duas abordagens principais podem ser utilizadas: o *backpressure*, que bloqueia temporariamente a inserção de novos eventos até que haja espaço disponível na fila; e o *blackout* (ou blecaute), que descarta os eventos recebidos quando a fila está cheia [Welsh et al. 2001].

2.3. Políticas de Processamento

O design flexível do SEDA permite que cada estágio defina sua própria política de processamento de eventos. Embora a política FIFO seja amplamente utilizada, outras estratégias podem ser mais adequadas dependendo do cenário e da carga de trabalho.

[1] `https://github.com/fernandoareias/erlda`

Essa flexibilidade possibilita otimizações significativas em sistemas de alta demanda, permitindo priorização inteligente de eventos ou ajuste dinâmico do comportamento [Cruz 2015; Gordon 2010].

2.4. Controlador

O controlador desempenha um papel estratégico na arquitetura SEDA. Ele deve evitar a alocação excessiva de *threads*, garantindo, ao mesmo tempo, que haja um número suficiente de *threads* para atender às demandas de concorrência de cada estágio [Welsh et al. 2001].

Diferentes estratégias de escalonamento podem ser implementadas dependendo das necessidades da aplicação. Por padrão, a arquitetura fornece uma estratégia baseada em heurística que monitora o comprimento da fila de entrada de cada estágio. Quando esse comprimento excede um limite predefinido, uma nova *thread* é adicionada ao estágio, respeitando o número máximo permitido. As *threads* ociosas por período especificado são removidas para liberar recursos computacionais [Welsh et al. 2001].

3. ERLDA

ERLDA é uma arquitetura inspirada no modelo LEDA, originalmente implementado em Lua [Salmito et al. 2013], mas adaptada para a linguagem Erlang. A escolha por Erlang fundamenta-se em suas propriedades fundamentais de concorrência verdadeira, isolamento entre processos e comunicação assíncrona, características essenciais para a construção de sistemas altamente concorrentes e resilientes [Armstrong 2003]. Essas funcionalidades permitem superar limitações encontradas em versões anteriores de SEDA [von Behren et al. 2003], especialmente no gerenciamento de concorrência e na resposta a falhas, simplificando a implementação de estágios e controladores.

3.1. Estágios

Visando aderir aos padrões da comunidade Erlang, implementamos um *behaviour* chamado *stage_behaviour*, responsável por abstrair a complexidade envolvida na criação de novos estágios. Esse *behaviour* define uma interface única, composta pelo seguinte *callback*:

```
1 -callback handle_command(Command :: term()) ->
2     {ok, Result :: term()} |
3     {forward, NextStagePid :: pid(), NewCommand :: term()} |
4     {error, Reason :: term()}.
```

Listing 1. Assinatura do callback handle_command/1.

Essa função é responsável por processar os comandos recebidos pelo estágio e retorna um dentre três possíveis resultados: {*ok, Result*}, indicando que o comando foi tratado com sucesso; {*error, Reason*}, caso ocorra algum problema durante o processamento; ou {*forward, NextStagePid, NewCommand*}, sinalizando que o comando deve ser repassado para outro estágio do fluxo. Essa definição flexível permite que cada estágio decida autonomamente se deve tratar localmente o comando ou encaminhá-lo para outro componente do sistema, possibilitando a construção de pipelines dinâmicos e modulares.

3.2. Workers

As *workers* no ERLDA são processos leves e dinamicamente escaláveis, responsáveis por executar os comandos recebidos pelos estágios. Assim como os *pools* de *threads* na arquitetura SEDA, elas permitem o processamento concorrente e assíncrono das tarefas.

Cada *worker* inicia sua execução dentro de um *loop* representado pela função *loop*/1. Nesse ciclo, ela aguarda receber um comando por meio da mensagem {*work, Command, From*}. Ao receber um comando, a *worker* delega seu processamento à função *handle_command/1* do módulo do estágio. Com base no resultado, realiza ações como registrar sucesso, tratar erros ou encaminhar o comando ao próximo estágio. Após finalizar, retorna ao início do *loop*, reiniciando o ciclo.

Essa estrutura permite que as *workers* fiquem ativos enquanto há demanda e voltem ao estado ocioso quando não há tarefas pendentes, otimizando o uso dos recursos do sistema. Além disso, o uso de mensagens assíncronas entre o estágio e suas *workers* oferece flexibilidade na comunicação e no encadeamento de estágios, reforçando a escalabilidade e resiliência do ERLDA.

3.3. Controlador

O controlador é implementado como um outro processo que verifica periodicamente (a cada 5 segundos) o tamanho da fila de mensagens do estágio (*QLen*) e compara com os limiares de escala (*scale_up_threshold* e *scale_down_threshold*). Se a fila exceder o limiar de aumento e houver espaço para novas *workers*, ele adiciona mais processos ao *pool*. Se a fila estiver abaixo do limiar de redução e houver *workers* em excesso, ele remove processos. O controlador também monitora a integridade do processo associado ao estágio, interrompendo suas operações caso o processo não esteja mais ativo. Essa abordagem garante uma alocação eficiente de recursos, adaptando-se dinamicamente às variações na carga de trabalho.

4. Estado da Arte

A Arquitetura Baseada em Estágios (SEDA), introduzida por Welsh et al., foi projetada para sistemas altamente concorrentes. Ela divide a aplicação em estágios interconectados por filas assíncronas, proporcionando isolamento entre componentes e permitindo o ajuste dinâmico de recursos. Essa abordagem controla eficientemente a sobrecarga e melhora a escalabilidade, sendo particularmente útil em ambientes sujeitos a picos de tráfego [Welsh et al. 2001].

Uma variação proposta por Gordon é voltada para servidores com alta demanda de CPU. Em vez de usar múltiplas *threads* por estágio, a sugestão é utilizar um único *thread* por núcleo físico, reduzindo o *overhead* e melhorando o desempenho em arquiteturas multicore [Gordon 2010].

A LEDA (*Lua Estaged Event-Driven Architecture*), desenvolvida por Salmito et al., foi inspirada na SEDA, mas introduz maior flexibilidade. A arquitetura utiliza conectores assíncronos entre estágios e permite particionar o sistema em *clusters*, facilitando a distribuição entre máquinas. Esse modelo aumenta a modularidade e a escalabilidade [Salmito et al. 2013].

Em trabalhos posteriores, Salmito, Moura e Rodriguez apresentaram uma versão baseada em grafos, onde os estágios são representados como nós interligados por canais

assíncronos. Essa abordagem possibilita diferentes configurações sem alterar a lógica do sistema, sendo ideal para servidores web escaláveis [Salmito et al. 2015].

A arquitetura LEDA, originalmente implementada em Lua, foi estendida por Cruz com a introdução de diferentes políticas de controle de recursos, visando otimizar o gerenciamento de *threads* e reduzir a ociosidade em cenários de carga variável. A abordagem demonstrou alta adaptabilidade e flexibilidade, especialmente em testes realizados em servidores HTTP, onde resultou em ganhos expressivos na eficiência do uso de recursos e na gestão de estágios de execução [Cruz 2015].

Com base nesses trabalhos, este artigo propõe o ERLDA, uma adaptação dos conceitos de SEDA e LEDA à linguagem Erlang. A escolha por Erlang se deve às suas vantagens naturais em concorrência, tolerância a falhas e comunicação assíncrona, características fundamentais para sistemas distribuídos modernos. O objetivo é explorar esses recursos para construir uma arquitetura modular, resiliente e altamente escalável.

5. Trabalhos Futuros

Pretendemos aplicar o ERLDA no desenvolvimento de um servidor web distribuído, utilizando exclusivamente as ferramentas nativas oferecidas pela plataforma Erlang. O objetivo principal é validar o desempenho, a escalabilidade e a eficiência dessa abordagem em cenários reais de uso. Esse estudo permitirá avaliar o comportamento do sistema sob diferentes cargas de trabalho, bem como testar diversas políticas de escalonamento dinâmico de recursos, contribuindo para uma compreensão mais profunda do potencial do ERLDA em ambientes distribuídos.

6. Conclusão

O ERLDA apresenta-se como uma proposta para a implementação de sistemas reativos e escaláveis, baseada em uma arquitetura inspirada no modelo SEDA, mas adaptada ao ecossistema e ao modelo de concorrência da linguagem Erlang. A utilização de processos leves, comunicação assíncrona e mecanismos nativos de monitoramento permite que o ERLDA gerencie fluxos de eventos de maneira eficiente e resiliente.

Em virtude dessas características, consideramos que o ERLDA se configura como uma base promissora para o desenvolvimento de aplicações orientadas a eventos, aproveitando plenamente as vantagens oferecidas pela plataforma Erlang/BEAM, tanto em ambientes locais quanto distribuídos.

References

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

Cruz, F. (2015). Uma interface de programação para controle de sobrecarga em arquiteturas baseadas em estágios. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio).

Gordon, M. E. (2010). Stage scheduling for cpu-intensive servers. Technical Report UCAM-CL-TR-781, University of Cambridge, Computer Laboratory. Technical report based on a dissertation submitted December 2009 for the degree of Doctor of Philosophy, University of Cambridge, Jesus College.

Salmito, T., Moura, A. L., and Rodriguez, N. (2015). A stepwise approach to developing staged applications. *J. Supercomput.*, 71(12):4663–4679.
Salmito, T., Moura, A. L. d., and Rodriguez, N. (2013). A flexible approach to staged events. In *Proceedings of the 2013 42nd International Conference on Parallel Processing*, ICPP '13, page 661–670, USA. IEEE Computer Society.
von Behren, R., Condit, J., and Brewer, E. (2003). Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, page 4, USA. USENIX Association.
Welsh, M., Culler, D., and Brewer, E. (2001). Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243.