Pontifícia Universidade Católica
do Rio de Janeiro

**Breno Riba da Costa Cruz**


# Uma interface de programação para controle de sobrecarga em arquiteturas baseadas em estágios

**Dissertação de Mestrado**

Dissertação apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do grau de Mestre em Informática.

Orientadora : Profª. Noemi Rodriguez
Co–Orientadora: Profª. Ana Lúcia de Moura

Rio de Janeiro
Fevereiro de 2015


PUC-Rio - Certificação Digital Nº 1212362/CA

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

**Breno Riba da Costa Cruz**

# Uma interface de programação para controle de sobrecarga em arquiteturas baseadas em estágios

Dissertação apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do grau de Mestre em Informática. Aprovada pela Comissão Examinadora abaixo assinada.

**Profª. Noemi Rodriguez**
Orientadora
Departamento de Informática — PUC–Rio

**Profª. Ana Lúcia de Moura**
Co–Orientadora
Departamento de Informática – PUC-Rio

**Prof. Markus Endler**
Departamento de Informática – PUC-Rio

**Profª. Silvana Rossetto**
Departamento de Ciência da Computação – UFRJ

**Prof. José Eugenio Leal**
Coordenador Setorial do Centro
Técnico Científico – PUC-Rio

Rio de Janeiro, 04 de Fevereiro de 2015

PUC-Rio - Certificação Digital Nº 1212362/CA

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

**Breno Riba da Costa Cruz**

Graduou–se em Ciência da Computação pela Universidade Gama Filho. Atualmente trabalha com Big Data atuando nas áreas de Sistemas Distribuídos e Aprendizado de Máquina na empresa BigData Corp.

PUC-Rio - Certificação Digital Nº 1212362/CA

Ficha Catalográfica

Riba, Breno

Uma interface de programação para controle de sobrecarga em arquiteturas baseadas em estágios / Breno Riba da Costa Cruz; orientador: Noemi Rodriguez; co–orientador: Ana Lúcia de Moura. — 2015.

75 f. : il. (color); 30 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2015.

Inclui bibliografia.

1. Informática – Teses. 2. Sistemas Distribuídos. 3. Modelos de Concorrência. 4. Arquiteturas Baseadas em Estágios. 5. SEDA. 6. Threads. 7. Eventos. I. Rodriguez, Noemi. II. Moura, Ana Lúcia de. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

# Agradecimentos

A Noemi e a Ana por toda orientação, dedicação, apoio, aprendizado e incentivo para a realização desse trabalho. Agradeço por todas as oportunidades que me deram durante esse trajeto.

Ao Tiago, que desenvolveu o LEDA e tornou esse projeto possível além da enorme ajuda com o entendimento e desenvolvimento do projeto.

Aos professores e funcionários do Departamento de Informática da PUC-Rio pelo grande aprendizado e auxílio.

A toda a equipe da empresa BigData Corp. por me incentivar e tornar essa jornada possível.

As amizades que tive a oportunidade de fazer desde o início do curso. Carol, Edu, Gi, Manu, Marcos, Paula e Rodrigo, obrigado por toda a experiência.

A minha família e a minha namorada Nathalia por me apoiarem durante todo o curso.

PUC-Rio - Certificação Digital Nº 1212362/CA

## Resumo

Riba, Breno; Rodriguez, Noemi; Moura, Ana Lúcia de. **Uma interface de programação para controle de sobrecarga em arquiteturas baseadas em estágios**. Rio de Janeiro, 2015. 75p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Controle de sobrecarga pode ser feito com o uso de políticas de escalonamento adequadas, que procuram ajustar dinamicamente os recursos alocados a uma aplicação. Pela dificuldade de implementação, muitas vezes desenvolvedores se veem obrigados a reprogramar o sistema para adequá-lo a uma determinada política. Através do estudo de diversas políticas de escalonamento, propomos neste trabalho um modelo de interface que permite a criação e monitoração de novas políticas dentro de arquiteturas baseadas em estágios. Implementamos a interface de programação proposta e exercitamos um conjunto de políticas que construímos sobre ela em duas aplicações com características de carga bem distintas.


PUC-Rio - Certificação Digital Nº 1212362/CA


## Palavras–chave

Sistemas Distribuídos; Modelos de Concorrência; Arquiteturas Baseadas em Estágios; SEDA; Threads; Eventos.

# Abstract

Riba, Breno; Rodriguez, Noemi (Advisor); Moura, Ana Lúcia de (Co-Advisor). **A programming interface for overload control in staged event based architectures**. Rio de Janeiro, 2015. 75p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Specific scheduling policies can be appropriate for overload control in different application scenarios. However, these policies are often difficult to implement, leading developers to reprogram entire systems in order to adapt them to a particular policy. Through the study of various scheduling policies, we propose an interface model that allows the programmer to integrate new policies and monitoring schemes to the same application in a Staged Event-Driven Architecture. We describe the implementation of the proposed interface and the results of it's use in implementing a set of scheduling policies for two applications with different load profiles.

PUC-Rio - Certificação Digital Nº 1212362/CA

## Keywords

Distributed Systems; Concurrent Models; Architectures Based on Stages; SEDA; Threads; Events.

# Sumário

1 Introdução **11**

2 Arquiteturas Baseadas em Estágios **14**
2.1 SEDA - *Staged Event-Driven Architecture* 15
2.2 LEDA - *Lua Event-Driven Architecture* 15
2.3 Discussão 18

3 Técnicas de controle de sobrecarga **19**
3.1 SEDA 20
3.2 Cohort 20
3.3 LRB 21
3.4 DBR 23
3.5 SRPT 23
3.6 Fila única de eventos 24
3.7 Workstealing 25
3.8 Discussão 25

4 Controladores **27**
4.1 Implementação de LEDA 28
4.2 SEDA 28
4.3 Cohort 32
4.4 LRB 34
4.5 DBR 38
4.6 SRPT 41
4.7 Fila única de eventos 44
4.8 Workstealing 44
4.9 Discussão 48

5 Aplicações **49**
5.1 Tratador de imagens 49
5.2 Servidor HTTP 52
5.3 Discussão 54

6 Conclusão **56**
6.1 Contribuições 56
6.2 Trabalhos futuros 57

7 Referências Bibliográficas **58**

A Modelo de interface **62**
A.1 Leda 63
A.2 Pool 65
A.3 Monitoring 67
A.4 Scheduler 70
A.5 Instances 73

PUC-Rio - Certificação Digital Nº 1212362/CA

PUC-Rio - Certificação Digital Nº 1212362/CA

# Lista de figuras


2.1 Projeto de um servidor no modelo *thread* por requisição 14
2.2 Estrutura de um servidor web de páginas estáticas (Salmito et al., 2013) 15
2.3 Componentes de um estágio na arquitetura SEDA (Salmito et al., 2013) 16
2.4 Etapas do processo desenvolvimento PCAM (Salmito et al., 2013) 17
2.5 Modelo das filas da arquitetura LEDA 18

3.1 Estrutura de um *array* com ordens de visitação 21

4.1 Funcionamento da arquitetura LEDA 29

5.1 Exemplo de Captchas utilizados para a binarização 49
5.2 Estágios do tratador de imagens 50
5.3 Tratamentos aplicados em uma imagem 51
5.4 Estágios do servidor HTTP 52


PUC-Rio - Certificação Digital Nº 1212362/CA

# Lista de tabelas

5.1 Tempo de processamento de 5.000 imagens em segundos por política 51
5.2 Tempo de requisições atentidas por usuário 53
5.3 Número de respostas por usuário por segundo 54

PUC-Rio - Certificação Digital Nº 1212362/CA

# 1 Introdução

Muitos sistemas que possuem uma variação grande de uso têm dificuldades para controlar sobrecarga de demanda em momentos de pico. Por conta disto, o tempo de resposta pode aumentar a níveis inaceitáveis para os usuários. No limite, variações de demanda podem até fazer com que o sistema fique fora do ar, refletindo em um péssimo serviço. Muitas empresas não são capazes de prover todos os recursos necessários para manter sistemas de larga escala com alta qualidade no ar, e além disso, em geral, aplicações não conseguem utilizar todo o potencial da máquina (Han et al., 2009).

Muitas vezes, desenvolvedores utilizam um dimensionamento estático de recursos em seus projetos como, por exemplo, um número fixo de threads. Uma implementação que realiza este tipo de controle pode ter problemas ao estipular limites muito baixos, já que o sistema pode assim subutilizar recursos. Por outro lado, limites muito altos podem sobrecarregar o sistema desnecessariamente. Uma questão que complica a escolha de configurações corretas é a dificuldade de experimentação. Muitas técnicas de controle de sobrecarga são estudadas apenas sob simulação (Kanodia e Knightly, 2000; Chen et al., 2001).

Controle de sobrecarga pode ser feito com o uso de políticas de escalonamento adequadas, que procuram ajustar dinamicamente os recursos disponíveis de acordo com a variação de demanda. A vantagem de criar políticas de ajuste dinâmico é que o sistema pode adotar comportamentos diferentes para evitar sua degradação. Nos momentos de pico, estas políticas podem priorizar determinadas requisições, bloquear novas requisições ou até mesmo redirecioná-las para outras máquinas caso a CPU esteja acima de um limite pré-definido pelo desenvolvedor. Muitos sistemas usam políticas que priorizam tarefas através de medições de CPU, memória e banda de rede (Welsh e Culler, 2003).

Modelos baseados em *threads* têm sido criticados por alguns autores por sua complexidade de uso (Ousterhout, 1996; Lee, 2006). Uma arquitetura que vem se destacando no cenário de servidores é a arquitetura baseada em estágios. Essa arquitetura combina *threads* e eventos, e tem como um de seus principais objetivos o suporte à alta demanda e concorrência. Na arquitetura de estágios,

PUC-Rio - Certificação Digital Nº 1212362/CA

uma aplicação é organizada como um pipeline de processamento e pode ser vista como um grafo direcionado onde os estágios são nós de processamento e as arestas representam filas de eventos. Cada estágio implementa parte da lógica da processamento e recebe eventos através de uma única fila de entrada.

Na proposta original de uma arquitetura de estágios, chamada SEDA (do inglês *Staged Event-Driven Architecture*) (Welsh et al., 2001), cada estágio possui seu próprio *pool* de *threads* e um *controlador* que ajusta os recursos do estágio de acordo com a sua demanda. Em seu trabalho, Welsh (Welsh et al., 2001) estudou possibilidades de recusar novas requisições, ou aumentar o número de *threads* para cada estágio em momentos de pico. Contudo, esse tipo de controle de recursos pode não atender a todo tipo de sistema. Por exemplo, uma aplicação com um grande número de estágios pode perder desempenho devido ao excesso no número de *threads* no sistema. Caso muitos estágios estejam com muitos eventos em suas filas de entrada, seus controladores irão criar novas *threads* até o seu limite e esta grande quantidade de *threads* pode causar a degradação do sistema.

Alguns trabalhos propuseram novos tipos de controle de sobrecarga para a arquitetura de estágios. Um trabalho que nos interessa é a pesquisa realizada por Gordon (Gordon, 2010). Em sua tese de doutorado, Gordon desenvolveu um conjunto de políticas de escalonamento para uma arquitetura de estágios, e analisou o desempenho de aplicações com características diferentes com o uso dessas políticas. Os resultados obtidos por Gordon mostram a influência das políticas de escalonamento e a dependência entre as aplicações e as políticas: aplicações com características diferentes podem responder de forma diversa a uma dada política de escalonamento. Apesar dos testes realizados com diversas políticas, Gordon fala da dificuldade de criação de novas políticas. Como cada política pode ter comportamentos muito diferentes, uma mesma aplicação acaba sofrendo muitas alterações para se adequar a uma determinada política. Em seu trabalho, Gordon precisou reprogramar suas aplicações para construir cada política.

Em nosso trabalho, utilizaremos a arquitetura LEDA (do inglês *Lua Event-Driven Architecture*) (Salmito et al., 2013), um modelo de extensão de SEDA que desacopla a especificação dos estágios dessa arquitetura a uma determinada configuração de execução. Nosso objetivo é permitir a criação e configuração de políticas de controle de sobrecarga capazes de atuar sobre um conjunto de estágios. A criação de novas políticas é um trabalho difícil e muitas vezes de alta complexidade. Sem um modelo de interface os desenvolvedores são obrigados a acessar complexos trechos de código em seus serviços para implementar diferentes políticas. O acesso direto ao código

PUC-Rio - Certificação Digital Nº 1212362/CA

torna o processo mais lento e abre brechas para a geração de erros. Para isso, investigamos os mecanismos e interfaces que uma arquitetura em estágios deve oferecer para permitir a construção de diferentes controles de sobrecarga, chamados de *controladores*, e implementamos alguns controladores para testar seu desempenho e eficácia em diferentes tipos de aplicações.

A interface disponibilizada pela arquitetura deve não só permitir que o desenvolvedor crie novas políticas de escalonamento, como também oferecer ferramentas necessárias para a monitoração do sistema. A monitoração do sistema ajuda o desenvolvedor a configurar o ambiente de execução de uma aplicação, permitindo a configuração da aplicação com o maior poder de processamento possível.

O restante deste trabalho está estruturado da seguinte forma: no capítulo 2 detalhamos as características de uma arquitetura baseada em estágios e a extensão dessa arquitetura chamada de LEDA. No capítulo 3, discutimos algumas técnicas de sobrecarga que podem ser aplicadas a uma arquitetura de estágios. No capítulo 4 demonstramos nossa investigação dos métodos necessários para que a interface de LEDA fosse estendida de forma a prover suporte para a construção de cada uma destas políticas. Ainda neste capítulo, detalhamos como cada política pode ser implementada com os métodos existentes e com os métodos que precisaram ser construídos. No capítulo 5 mostramos os testes feitos com diversas políticas em duas diferentes aplicações e por fim, no capítulo 6 a conclusão do trabalho.

PUC-Rio - Certificação Digital Nº 1212362/CA

# 2
# Arquiteturas Baseadas em Estágios

O conceito de estágios foi introduzido na arquitetura SEDA (Welsh et al., 2001) para permitir a criação de servidores altamente concorrentes. Uma maneira simples e comum de implementar servidores fora das arquiteturas baseadas em estágios é através do modelo *thread* por requisição (Schmidt et al., 2000). Como ilustrado na figura 2.1, cada requisição é associada a uma *thread*. Cada *thread* realiza o processamento de uma requisição e retorna o resultado para o cliente. Apesar de relativamente simples de se desenvolver, este modelo não é escalável. O aumento no número de *threads* pode levar a gargalos de desempenho e sobrecarga devido aos custos das travas e de mecanismos de sincronização.

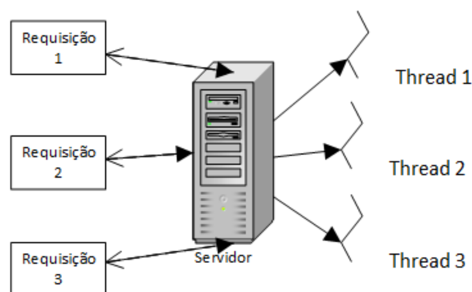


Figura 2.1: Projeto de um servidor no modelo *thread* por requisição

Para evitar o uso excessivo de recursos, alguns desenvolvedores adotam sistemas com um limite máximo no número de *threads* que podem ser criadas. Quando o número máximo é atingido, novas conexões são descartadas. Essa estratégia é utilizada pelo Apache (Behlendorf et al., 1995) e pelo IIS (Microsoft, 2011). Limitando o número de *threads*, o servidor evita a degradação do sistema, porém pode ficar inacessível para diversos usuários em momento de pico. Welsh propôs como alternativa a arquitetura SEDA, baseada em estágios. A arquitetura SEDA foi criada para lidar com alta demanda e permitir a construção de aplicações concorrentes de maneira simples através do conceito de estágios concorrentes.

Na primeira seção deste capítulo explicaremos o funcionamento das arquiteturas baseadas em estágios. Na seção seguinte, com base na análise das características de SEDA, explicaremos o funcionamento da arquitetura

PUC-Rio - Certificação Digital Nº 1212362/CA

LEDA, o modelo de extensão de SEDA que utilizamos para a construção de nossa interface.

## 2.1 SEDA - Staged Event-Driven Architecture

Esta arquitetura segue um modelo de concorrência híbrido, que combina eventos e threads (Welsh et al., 2001). Uma aplicação em SEDA é constituída de estágios, onde cada estágio representa uma parte do problema. Cada estágio possui uma fila de eventos, que é o único meio de comunicação entre os estágios da aplicação. Uma aplicação criada nesta arquitetura pode ser representada como um grafo orientado, conforme ilustrado na figura 2.1.

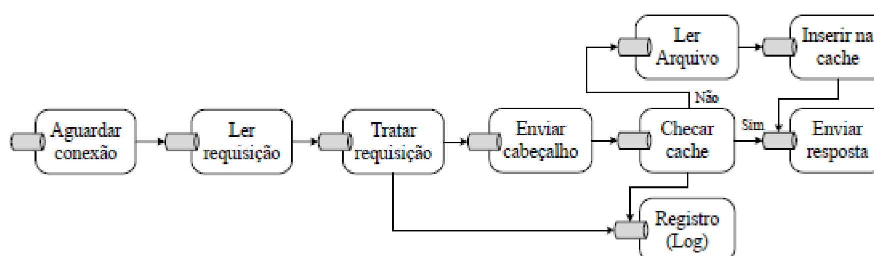


Figura 2.2: Estrutura de um servidor web de páginas estáticas (Salmito et al., 2013)

Um estágio consiste em um *pool* de *threads*, uma fila de entrada, um tratador de eventos e um controlador conforme ilustrado na figura 2.3. Cada estágio possui um pequeno número de *threads* em seu *pool* de *threads*. Cada *thread* remove repetidamente eventos da fila de entrada de seu estágio e executa instâncias do tratador de eventos. Cada evento processado pode ou não gerar novos eventos, que serão encaminhados para a fila de eventos dos demais estágios da aplicação. O controlador de cada estágio atua de forma independente, pois cada controlador controla apenas os recursos de seu estágio. Cada controlador é capaz de ajustar os recursos do estágio dinâmicamente, podendo criar ou remover *threads* do *pool* de seu estágio.

## 2.2 LEDA - Lua Event-Driven Architecture

LEDA (Salmito et al., 2013) é uma extensão do modelo orientado a estágios feita em Lua (Ierusalimschy, 2013) para aplicações Lua com o objetivo de incentivar o desacoplamento entre os estágios da aplicação. O desacoplamento entre os estágios permite a construção de um sistema mais flexível. LEDA permite que sistemas possam ser configurados de diversas maneiras, podendo se adaptar a diferentes ambientes de execução. Esta extensão permite

PUC-Rio - Certificação Digital Nº 1212362/CA

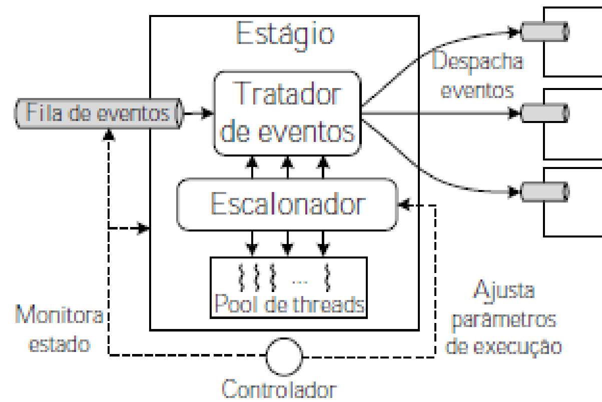


Figura 2.3: Componentes de um estágio na arquitetura SEDA (Salmito et al., 2013)

que o desenvolvedor configure o seu sistema de acordo com suas necessidades, como por exemplo, utilizar um mesmo *pool* de *threads* para um conjunto de estágios, chamados de aglomerado. Aplicar um mesmo *pool* de *threads* para um aglomerado permite criar aplicações com maior número de estágios e ainda diminuir a chance de uma *thread* ficar ociosa, pois as *threads* de um aglomerado só estarão ociosas caso todos os estágios estejam sem eventos em sua fila de eventos. Outra vantagem desta arquitetura é que um controlador é capaz de monitorar todos os estágios do aglomerado a que pertence, já que um controlador não faz parte individualmente de um estágio na arquitetura LEDA. Em SEDA, a oscilação no número de threads em todos os estágios pode gerar problemas de desempenho (Gordon, 2010). Se todos os estágios estiverem com alta demanda, todos os controladores irão aumentar o número de *threads* do *pool* de seu estágio podendo gerar problemas de desempenho para a aplicação.

O modelo de programação da arquitetura LEDA segue as etapas da metodologia PCAM, proposta por Foster (Foster, 1995) para construir sistemas paralelos. A figura 2.4 contém uma representação das etapas desta metodologia.

A metodologia PCAM, adaptada para LEDA, consiste em quatro etapas:

1. *Particionamento ou decomposição*
   Esta etapa é responsável por decompor o problema em subproblemas independentes, ou seja, decompor uma aplicação em estágios.

2. *Comunicação*
   Esta etapa é responsável por criar o fluxo de informações, ou seja, configurar quais são os estágios que se comunicam.

PUC-Rio - Certificação Digital Nº 1212362/CA

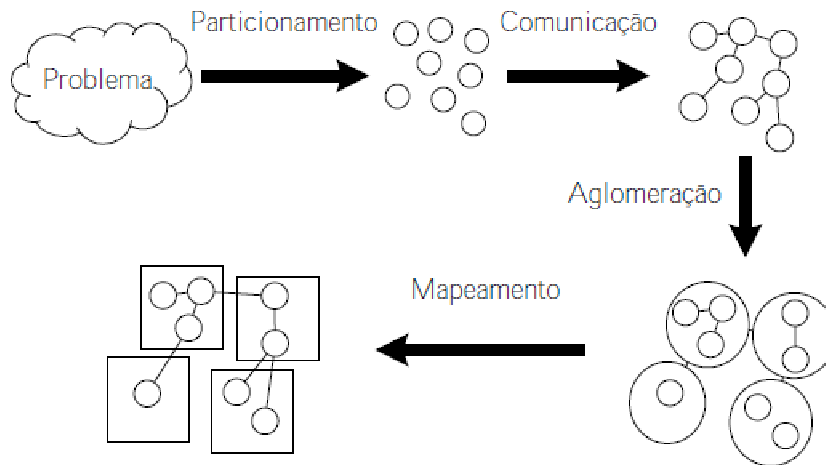


Figura 2.4: Etapas do processo desenvolvimento PCAM (Salmito et al., 2013)

3. *Aglomeração*
   Esta etapa é responsável por aglomerar os estágios. Um aglomerado é o nome dado a um conjunto de estágios que compartilham recursos.

4. *Mapeamento*
   Esta etapa é responsável por mapear os aglomerados da etapa anterior em processos. Cada processo pode ou não estar na mesma máquina.

PUC-Rio - Certificação Digital Nº 1212362/CA

Em LEDA, cada estágio possui sua fila de instâncias, sua fila de eventos e seu tratador de eventos. As instâncias de um estágio possuem estado próprio e são criadas para permitir a execução de eventos em paralelo. Cada instância é executada por uma *thread* do *pool* para processar os eventos da fila de eventos do estágio. O número de instâncias representa o número máximo de eventos que um estágio pode executar paralelamente. Podemos ter um estágio com duas instâncias em sua fila de instâncias e dez eventos em sua fila de entrada, porém ele só será capaz de executar dois eventos por vez. Esta técnica permite que estágios com maior prioridade possam ter mais instâncias e por consequência executar mais eventos do que os demais.

Cada aglomerado, ou seja, cada conjunto de estágios compartilha uma fila de processamento global e um *pool* de *threads*. Quando um evento chega na fila de eventos de um estágio, este evento é combinado com uma instância e esta combinação é colocada na fila de processamento global, que é a única estrutura de dados acessíveis pelas *threads*. As *threads* ficam em *loop* dentro da fila de processamento do aglomerado para processar novos eventos. Quando uma *thread* termina o processamento de um evento, a instância é novamente retornada para a fila de instâncias de seu estágio e então novamente combinada com um novo evento. Caso o estágio não possua instâncias livres no momento

da chegada de um novo evento, este evento é acumulado na fila de eventos do estágio. A figura 2.5 contém uma representação das filas da arquitetura LEDA.

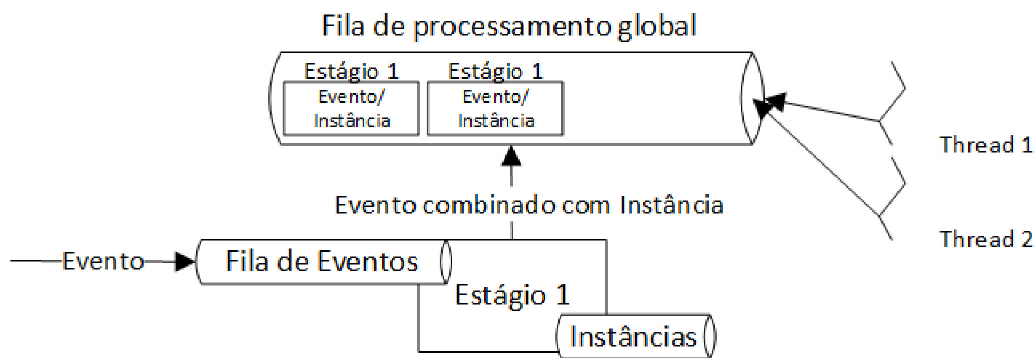


Figura 2.5: Modelo das filas da arquitetura LEDA

## 2.3 Discussão

Neste capítulo apresentamos o funcionamento das arquiteturas baseadas em estágios e da arquitetura LEDA, uma extensão da arquitetura SEDA que promove a dissociação entre a lógica da aplicação e sua estrutura de execução.

Em LEDA, é possível dissociar componentes de um estágio, como por exemplo aplicar um mesmo *pool* a um aglomerado. A flexibilidade implementada em LEDA facilita a configuração das aplicações, permitindo um maior controle para a aplicação.

PUC-Rio - Certificação Digital Nº 1212362/CA

# 3
# Técnicas de controle de sobrecarga

Por conta da degradação de sistemas em momentos de pico, técnicas de controle de sobrecarga vêm sendo exploradas por diversos autores para garantir um desempenho aceitável das aplicações face à variação em sua demanda (Zeldovich et al., 2003; Han et al., 2009; Gordon, 2010). O objetivo destas técnicas é permitir que um sistema possa distribuir adequadamente os recursos disponíveis em momentos de alta demanda.

Muitos autores investem em técnicas do tipo Mestre-Trabalhador e *Workstealing* para balancear o processamento do sistema. Nas arquiteturas do tipo mestre-trabalhador, o mestre normalmente inicializa, distribui e coleta trabalho entre os nós classificados como trabalhadores. Os nós trabalhadores processam as requisições e retornam os dados para o mestre (Tanese, 1989). Nas arquiteturas do tipo *Workstealing*, processos podem capturar requisições de outros processos que estiverem com maior demanda (Burton e Sleep, 1981).

Outros autores trabalham buscando técnicas de escalonamento que evitem que o desempenho do sistema sofra muita degradação diante de picos de carga. No contexto de estágios, um trabalho que nos interessa é o realizado por Gordon (Gordon, 2010). Nesse trabalho, Gordon propôs e analisou algumas técnicas de controle de sobrecarga. Diferentemente da arquitetura original de SEDA, que define o uso de um pool de threads exclusivo por estágio, as políticas de escalonamento analisadas por Gordon utilizam um único pool de threads, dimensionado de acordo com o número de CPU's através do modelo *thread* por *CPU* [1].

Neste capítulo iremos levantar um conjunto de técnicas de controle de sobrecarga aplicáveis a uma arquitetura de estágios para que possamos investigar a interface necessária para a implementação de uma gama variada de tais técnicas. As políticas *DBR*, *SRPT* e *Color* foram criadas por Gordon (Gordon, 2010) inspiradas no trabalho de outros autores, como por exemplo o trabalho de Boxma (Boxma et al., 1993), que explora formas eficientes de visitação entre filas de requisições. Falaremos também sobre técnicas criadas e exploradas por outros autores dentro e fora das arquiteturas de estágios como

[1] Modelo onde o número de threads é proporcional ao número de CPU's

PUC-Rio - Certificação Digital Nº 1212362/CA

forma de controle de sobrecarga. Através do estudo dessas diversas técnicas é possível estabelecer os métodos que devem ser expostos pela interface de LEDA para permitir a criação de variadas políticas.

## 3.1 SEDA

Apesar da arquitetura SEDA ter sido criada para ser utilizada em sistemas que demandam alta concorrência, a distribuição de recursos da arquitetura original pode não ser eficaz para todo tipo de aplicação. A proposta inicial de SEDA associa cada estágio a uma fila de entrada e um *pool* com um número fixo de *threads*. As *threads* do *pool* visitam apenas a fila de entrada do seu estágio. Por conta desta limitação, estágios sem requisições continuam com seu número de *threads* inicial, alocando recursos da máquina de forma desnecessária. A proposta de um número fixo de *threads* por estágio dificulta a escolha do dimensionamento adequado no número de *threads* para cada aplicação.

Alguns autores exploraram técnicas de ajuste dinâmico de recursos, tornando possível a um controlador ajustar o número de *threads* no *pool* do estágio a que pertence (Welsh et al., 2001; Li et al., 2006). Quando a fila de entrada de um estágio excede um limite de tamanho, o controlador do estágio aloca mais *threads* em seu *pool* até um limite máximo. *Threads* são removidas do *pool* de *threads* quando o estágio permanece sem requisições por um período de tempo. Além deste controle, Welsh (Welsh e Culler, 2003) também explorou algumas técnicas de rejeição e redirecionamento de novas requisições e até mesmo técnicas de redução na qualidade do serviço prestado em momentos de pico.

PUC-Rio - Certificação Digital Nº 1212362/CA

## 3.2 Cohort

*Cohort* foi a primeira técnica de controle de sobrecarga proposta para a arquitetura de estágios baseada em uma política de escalonamento *thread por CPU* (Larus e Parkes, 2001). Esta técnica reúne todas as *threads* disponíveis reunidas em um único *pool* compartilhado entre todos os estágios. A idéia básica é que as *threads* alocadas neste *pool visitem* os estágios da aplicação e, em cada visita, processem os eventos presentes na fila de entrada do estágio visitado. A política de *Cohort* foi criada visando explorar a localidade de código e dados. Para isso, a ordem de visitação dos estágios é definida da seguinte forma: inicialmente, todas as *threads* iteram dentro da fila de eventos do primeiro estágio da aplicação. Quando uma *thread* não consegue capturar

eventos para processar, ou seja, a fila do estágio está vazia, esta *thread* visita o próximo estágio e estes passos se repetem até que a *thread* chegue no último estágio. Quando a *thread* chega no último estágio o mesmo processo é feito de trás para frente. A idéia de visitar os estágios nesta ordem foi inspirada pela observação que normalmente o grafo da aplicação resulta em um pipeline, com o tratamento de requisições iniciando-se no primeiro estágio.

Como cada *thread* só passa para o estágio seguinte caso a fila de eventos do estágio visitado estiver vazia, este controle pode causar um aumento no tempo de resposta caso o sistema esteja sobrecarregado. Se novos eventos não pararem de chegar ao estágio onde as *threads* estão processando eventos, as *threads* não conseguirão passar para o estágio seguinte até processar todos os eventos. Neste caso, os demais estágios ficarão sem *threads* para continuar seu processamento até que o estágio sendo visitado não receba mais eventos. Para evitar o aumento no tempo de resposta, Harizopoulos (Harizopoulos e Ailamaki, 2002) apresentou uma alternativa ao modelo original de *Cohort*, configurando um limite máximo de eventos que as *threads* poderiam processar para cada estágio. Desta forma, quando este limite fosse atingido, as *threads* alocadas naquele estágio seriam alocadas para o estágio seguinte independentemente dos novos eventos que estivessem chegando.

PUC-Rio - Certificação Digital Nº 1212362/CA

## 3.3 LRB

Criamos a política LRB (do inglês *Load Rate Based*) inspirada na técnica MG1 criada por Gordon. A técnica MG1 foi inspirada no trabalho de Boxma (Boxma et al., 1993), que analisou formas de visitação eficientes entre filas de requisições. A idéia da técnica explorada por Boxma é que uma *thread* possa visitar filas contendo requisições em uma certa ordem de forma circular. O resultado é um *array* que define a ordem de visitação das filas de requisições, onde cada posição do *array* representa uma fila a ser visitada conforme ilustrado na figura 3.1.

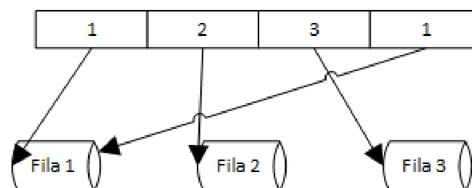


Figura 3.1: Estrutura de um *array* com ordens de visitação

A maneira com que as *threads* se comportam para atender as requisições segue alguns parâmetros:

1. *Ordem de visitação*
   Boxma apresentou diversas maneiras de criar uma ordem de visitação. Uma delas consiste em iniciar da primeira fila e passar a *thread* para a fila seguinte até chegar na última fila, reiniciando o processo pela primeira fila de requisições. Uma outra consiste em iniciar da primeira fila e seguir até a última delas. Em seguida a *thread* retorna para a fila anterior à última até chegar novamente na primeira fila e reiniciar o processo.

2. *Comportamento das threads*
   No primeiro modelo as *threads* executam as requisições da fila em que estão alocadas até que não existam mais requisições para processar. No segundo modelo, as *threads* executam todas as requisições que estiverem na fila no momento da visita, ignorando as requisições que chegaram durante este processo. Por fim, o último modelo processa as requisições até um limite pré-configurado pelo desenvolvedor. Caso a fila já esteja sem requisições no momento em que as *threads* são alocadas, todas as *threads* são realocadas para a fila seguinte de acordo com a ordem estabelecida.

PUC-Rio - Certificação Digital Nº 1212362/CA

Na política LRB, cada fila passa a ser a fila de entrada de cada estágio. Esta política é baseada na idéia de que estágios com maior demanda devem receber mais visitas.

A demanda de cada estágio é medida através da quantidade de eventos que chegaram na fila de eventos dos estágios desde a última medição. Esta política utiliza um mesmo *pool* de *threads* para todos os estágios da aplicação. Os estágios são visitados pelas *threads* de acordo com a ordem criada no momento da construção da ordem de visitação. Inicialmente, a ordem de visitação das *threads* segue a ordem do próprio pipeline. O *array* é reconstruído de acordo com a demanda de cada estágio quando o último estágio da ordenação recebe uma visita das *threads*. Utilizamos um *array* de tamanho mínimo 2 vezes maior do que o número total de estágios na aplicação. Este tamanho permite que alguns estágios possam ser visitados mais de uma vez.

A frequência com que cada estágio é visitado é calculada em dois passos. Primeiro é calculado a porcentagem dos eventos que chegaram para o determinado estágio em comparação com o total de eventos de entrada de todos os estágios do aglomerado. Com a porcentagem de demanda para cada estágio, calculamos a porcentagem deste total com o total de posições disponíveis no *array*, onde cada resultado é o mais próximo de um inteiro arredondado para cima. Por exemplo: caso tenhamos uma aplicação com 3 estágios, podemos utilizar um *array* de tamanho 9 para definir o total de vezes que cada estágio

deverá ser visitado até que o *array* seja novamente reconstruído. Supondo que a demanda dos estágios 1, 2 e 3 seja 50%, 30%, 20% respectivamente, os estágios apareceriam 5, 3 e 2 vezes respectivamente. O *array* resultante para esta etapa seria 1,2,1,2,1,2,1,3,1,3 onde cada número indica o número do estágio a ser visitado.

Visitar estágios de acordo com a sua demanda possui duas vantagens: evita que as *threads* visitem com frequência os estágios que estiverem com pouca ou nenhuma demanda e garante que os estágios com maior demanda sejam visitados mais frequentemente, diminuindo assim o tempo de processamento.

## 3.4 DBR

A técnica DBR (do inglês, *Drum Buffer Rope*) (Gordon, 2010) foi inspirada pela metodologia criada por Goldratt (Goldratt, 1990). Essa metodologia foi criada para regular o fluxo de um processo de trabalho através de uma linha de produção de forma que o processo possa sempre funcionar em sua capacidade máxima. Para isso, as tarefas são ordenadas pelas horas de trabalho necessárias para concluí-las e são priorizadas em função do tempo previsto para sua conclusão, isto é, as tarefas consideradas mais lentas são realizadas primeiro.

Na aplicação desta técnica à arquitetura de estágios, os estágios são ordenados de acordo com sua taxa de serviço. Estágios com menor taxa de serviço são visitados primeiro. Esta técnica utiliza um mesmo *pool* de *threads* para todos os estágios da aplicação. Quando uma *thread* visita um estágio com sucesso, ou seja, processou pelo menos um evento durante a visita, esta *thread* retorna ao primeiro estágio da ordenação. A taxa de serviço dos estágios é recalculada após a primeira *thread* chegar no último estágio da ordenação. Esta técnica foca nos gargalos da aplicação para evitar uma possível degradação do sistema.

## 3.5 SRPT

Esta técnica foi inspirada pelo trabalho de Schrage (Schrage e Miller, 1966), que criou a técnica chamada de SRPT (do inglês, *Shortest Remaining Processing Time*). Inicialmente uma fila de prioridades é criada. A prioridade de cada requisição depende do seu tempo de processamento, e o tempo de processamento de cada requisição é conhecido no momento em que ela chega na fila de prioridades. As requisições que necessitam de um tempo

PUC-Rio - Certificação Digital Nº 1212362/CA

de processamento menor são processadas primeiro, ou seja, possuem maior prioridade.

Uma requisição pode ser preemptada caso uma nova requisição chegue com um tempo de processamento menor do que o tempo restante para processar a requisição atual. A idéia da técnica criada por Schrage é priorizar o tratamento de tarefas mais próximas de serem finalizadas. Apesar de ser uma técnica muito antiga, ainda é bastante utilizada (Harchol-Balter et al., 2001; Harchol-Balter et al., 2003; Gong e Williamson, 2003; Gordon, 2010).

A adaptação feita por Gordon (Gordon, 2010) para integrar esta técnica às arquiteturas baseadas em estágios foi priorizar estágios que estão no fim do pipeline. Priorizando estágios finais da aplicação, as requisições que estiverem mais próximas de serem finalizadas serão atendidas mais rapidamente. Na política SRPT, um mesmo *pool* de *threads* é compartilhado entre os estágios e cada estágio possui sua própria fila de entrada. As *threads* visitam os estágios de trás para frente, começando no último estágio do pipeline. Para cada estágio visitado com sucesso, ou seja, com pelo menos uma requisição processada por uma das *threads*, as *threads* iniciam o processo novamente no fim do pipeline. Esta política visa diminuir o tempo de resposta e maximizar o *throughput* da aplicação.

PUC-Rio - Certificação Digital Nº 1212362/CA

## 3.6 Fila única de eventos

A última política explorada por Gordon foi a utilização de uma fila única de eventos. Gordon chamou essa política de *Color* em referência ao trabalho de Zeldovich (Zeldovich et al., 2003). No trabalho de Zeldovich, cada estágio da aplicação é associado a uma cor. Estágios da mesma cor são processados serialmente e estágios de cores diferentes podem ser processados em paralelo.

Na adaptação criada por Gordon (Gordon, 2010), os estágios deixam de possuir uma fila exclusiva de eventos. Todos os estágios da aplicação estão conectados a uma fila de processamento global, onde cada evento é associado a uma cor. A cor de cada evento é um identificador único que serve para indicar a que estágio o evento pertence. A vantagem desta política é que estágios sem requisições nunca serão visitados, já que as *threads* do sistema estão iterando apenas na fila de processamento global. Esta idéia de uma fila única é utilizada por grandes serviços para construir aplicações altamente escaláveis (Amazon, 2006).

Uma outra adaptação com o uso de uma fila única de eventos seria a aplicação da técnica chamada Mestre-trabalhador (Freeman et al., 1999). Este tipo de técnica funciona da seguinte maneira: o nó mestre inicializa a

aplicação, divide o problema em partes e as envia para os nós classificados como trabalhadores. Cada nó trabalhador executa sua tarefa e retorna para o nó mestre. Por fim, o nó mestre combina todos os resultados retornados pelos nós trabalhadores e termina sua execução. Esta técnica é utilizada em larga escala com técnicas de MapReduce e é aplicada dentro do Hadoop, um framework para processamento distribuído (Dean e Ghemawat, 2008; Foundation, 2006).

Através do uso de uma única fila de eventos, cada parte do problema representa um evento na fila única e cada *thread* representa um nó trabalhador. Estes eventos são capturados pelas *threads* e então processados. Desta forma, a técnica Mestre-trabalhador pode ser adaptada para as arquiteturas baseadas em estágios.

## 3.7 Workstealing

Esta técnica foi proposta inicialmente para as arquiteturas baseadas em *threads* mas pode ser aplicada dentro de arquiteturas baseadas em estágios. Técnicas de *Workstealing* permitem que os nós de processamento possam *roubar* trabalho de outros nós que estejam em sua capacidade máxima (Blumofe e Leiserson, 1999). Esta técnica é utilizada para balancear o processamento do sistema. Em seu trabalho, Gaud (Gaud et al., 2010) explora está técnica criando uma fila de requisições e uma *thread* para cada CPU da máquina. Quando uma fila está sem requisições para processar, uma *thread* pode então roubar blocos de requisições de outras filas.

Esta técnica pode ser aplicada dentro das arquiteturas baseadas em estágios através do roubo de recursos com a configuração de um *pool* por estágio. Como cada estágio possui seu próprio tratator de eventos, um estágio não pode roubar eventos de outro estágio pois isto obrigaria que todos os estágios tivessem acesso ao tratador de eventos dos demais. Na adaptação desta técnica para as arquiteturas de estágios, cada estágio é configurado com seu próprio *pool* de *threads*. Se algum estágio estiver com *threads* ociosas, um controlador é capaz de remover algumas *threads* do *pool* deste estágio e alocá-las em outro *pool* de um estágio com maior demanda.

## 3.8 Discussão

Neste capítulo vimos algumas técnicas de controle de desempenho que foram criadas inicialmente para as arquiteturas de estágios e técnicas criadas para outras arquiteturas que podem ser adaptadas e aplicadas às arquiteturas de estágios. A importância da construção de novas técnicas deve-se ao fato

PUC-Rio - Certificação Digital Nº 1212362/CA

de que diferentes aplicações podem responder de forma diversa a uma dada técnica de controle.

O estudo de novas técnicas de controle nos permite analisar o que a interface de LEDA precisa expor para permitir a criação de controladores com diferentes comportamentos. No capítulo seguinte, demonstraremos como cada uma destas técnicas podem ser implementadas utilizando os métodos expostos pela interface criada em LEDA.

PUC-Rio - Certificação Digital Nº 1212362/CA

# 4
# Controladores

A partir do estudo de técnicas de controle de sobrecarga feito no capítulo anterior, fizemos um levantamento de requisitos para analisar quais métodos a interface de LEDA deve expor para permitir a construção de controladores com diversos tipos de comportamento. Um controlador é responsável por monitorar e alocar recursos para um conjunto de estágios. Com o uso da nova interface, desenvolvedores podem aplicar diferentes técnicas de controle de sobrecarga a diferentes tipos de aplicação e ambientes de execução de forma simples, sem que seja necessário entender o código por trás da arquitetura. Neste capítulo iremos demonstrar como as técnicas descritas no capítulo anterior podem ser adaptadas e implementadas utilizando a interface criada em LEDA.

Alguns dos métodos necessários já existiam em LEDA. Neste trabalho, organizamos estes métodos em classes e criamos as demais funções identificadas como necessárias. A interface resultante é dividida em 5 classes: *Leda*, *Monitoring*, *Scheduler*, *Pool* e *Instances*. *Leda* engloba os métodos globais. *Monitoring* reúne os métodos utilizados para a monitoração do sistema, como por exemplo *callbacks* de *timer* e captura de *throughput* por estágio. *Scheduler* reúne métodos de priorização de estágios e métodos de configuração da ordem de visitação das *threads*. *Pool* reúne métodos para a criação de *pools* e criação e remoção de novas *threads*. Por fim, *Instances* reúne métodos de configuração de instâncias para cada estágio, como por exemplo criação de novas instâncias ou captura do número de instâncias de um determinado estágio. O apêndice A contém a descrição completa da interface.

Existem algumas diferenças entre a arquitetura SEDA e a arquitetura LEDA. Em SEDA, cada estágio possui seu próprio *pool* de *threads*, sua fila de entrada e seu próprio controlador. Cada *thread* do *pool* de um estágio processa eventos da fila do estágio a que está associada. O controlador é o responsável por monitorar o estágio, podendo modificar o número de *threads* de seu *pool* e até mesmo descartar novos eventos em caso de sobrecarga. Na arquitetura LEDA, um controlador e um *pool* de *threads* são compartilhados entre os estágios de um aglomerado. Um aglomerado é um conjunto de estágios executados em um mesmo processo. O controlador é capaz de monitorar os

PUC-Rio - Certificação Digital Nº 1212362/CA

recursos de um aglomerado ao invés de controlar apenas um estágio como o que ocorre em SEDA.

A seguir, demonstraremos como as técnicas descritas no capítulo anterior podem ser criadas com o uso da interface de LEDA. Em cada técnica, falaremos quais os métodos necessários para sua construção e como fizemos para construir o seu comportamento. Para poder explicar a implementação de cada técnica, inicialmente descreveremos brevemente a implementação de LEDA.

## 4.1 Implementação de LEDA

A figura 4.1 ilustra a implementação de LEDA. Em LEDA, cada aglomerado possui seu próprio controlador e seu próprio *pool* de *threads*, e cada estágio possui uma fila de eventos e uma fila de instâncias. Além das filas de eventos e de instâncias de cada estágio, LEDA implementa uma fila de processamento global, descrita mais adiante. Em SEDA, um único estado global é compartilhado entre todas as *threads* de um estágio. Isso obriga que os estágios sincronizem o acesso a dados. Em LEDA, as instâncias de um estágio correspondem a *threads* de nível de aplicação, e cada instância tem seu próprio estado global. Isto permite com que os estágios não precisem lidar com sincronização de dados, já que não existe alteração simultânea de informações. A fila de processamento global é uma fila de instâncias associadas a eventos específicos. Ela é ordenada pela prioridade do estágio associado a cada instância. As *threads* do *pool* de cada estágio continuamente removem as instâncias da fila de processamento global e as executam, para processar o evento associado a cada instância. Portanto, o número de instâncias em um estágio indica o grau máximo de paralelismo que o estágio pode obter, por exemplo: um estágio com 2 instâncias só poderá ter 2 *threads* processando seus eventos ao mesmo tempo.

A fila de eventos de cada estágio acumula eventos quando não há instâncias livres para tratá-los. Quando um novo evento chega a um estágio, caso haja uma instância livre na fila de instâncias, este evento é associado com uma instância livre e colocado na fila global. Caso o estágio não tenha instâncias livres, o evento é colocado na fila de eventos do estágio e aí permanece até que uma instância termine seu processamento e possa ser associada a um novo evento.

## 4.2 SEDA

Conforme discutido no capítulo anterior, na arquitetura SEDA cada estágio é associado a um *pool* de *threads* diferente. Na proposta inicial de Welsh,

PUC-Rio - Certificação Digital Nº 1212362/CA

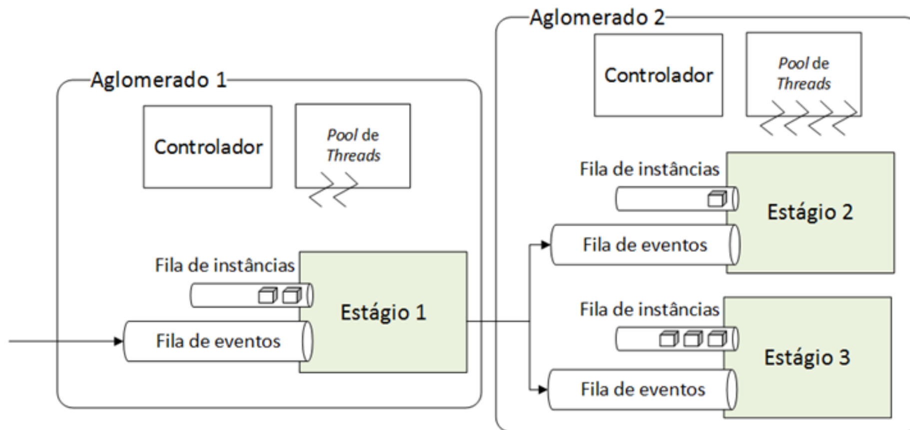


Figura 4.1: Funcionamento da arquitetura LEDA

cada *pool* recebe um número fixo de *threads*. Propostas mais recentes tratam o número de *threads* de forma dinâmica para acompanhar a demanda do sistema. Welsh também permitiu a rejeição de novos eventos por determinados estágios caso um limite de eventos fosse atingido. Para construir um controlador onde cada estágio possua seu próprio *pool* de *threads* com ajuste dinâmico, precisamos disponibilizar os seguintes métodos em nossa interface:

1. Permitir a criação de *pools* de *threads* associados a estágios individuais.
2. Permitir a criação e remoção de novas *threads* para balancear os recursos da aplicação.
3. Permitir a captura da quantidade de eventos e de instâncias de cada estágio para que seja possível monitorar a demanda de cada estágio.
4. Permitir habilitar e desabilitar a fila de eventos de cada estágio para descarte de novos eventos.
5. Permitir a criação de *callbacks* que possam ser disparadas de tempos em tempos para que o sistema possa ser monitorado constantemente.

O trecho de código abaixo demonstra um exemplo de um controlador SEDA com monitoração dinâmica:

```
1 local leda = require 'leda'
2
3 -- Creating a stage
4 local stage1=leda.stage(
5    function()
6       print("Hello World!")
7    end)
```

PUC-Rio - Certificação Digital Nº 1212362/CA

```
8
9 local stage2=...
10 local stage3=...
11
12 -- Creating 10 free instances for each stage
13 leda.instances.addInstances(stage1,10)
14 leda.instances.addInstances(stage2,10)
15 leda.instances.addInstances(stage3,10)
16
17 -- Configuring queue capacity to a maximum number of 100 events
18 leda.setQueueCapacity(stage1,100)
19 leda.setQueueCapacity(stage2,100)
20 leda.setQueueCapacity(stage3,100)
21
22 -- Creating new thread pool with 2 threads
23 -- and associating with each stage
24 local pool1 = leda.pool.new()
25 leda.pool.add(pool1,2)
26 leda.pool.setPool(stage1,pool1)
27
28 local pool2 = leda.pool.new()
29 leda.pool.add(pool2,2)
30 leda.pool.setPool(stage2,pool2)
31
32 local pool3 = leda.pool.new()
33 leda.pool.add(pool3,2)
34 leda.pool.setPool(stage3,pool3)
35
36 local stages = {stage1,stage2,stage3}
37
38 local onTimer = function ()
39    for i,stage in ipairs(stages) do
40       -- Calculating events count
41       local queueSize = leda.monitoring.getQueueSize(stage)
42       local freeInstances = leda.instances.getFreeInstancesCount(
            stage)
43       local instancesCount = leda.instances.getInstancesCount(
            stage)
44       local eventsCount = queueSize + (instancesCount -
            freeInstances)
45
46       -- Get stage pool threads count
47          local pool = leda.pool.getPool (stage)
48        local threadsCount = leda.pool.getThreadsCount(pool)
49
50       if (eventsCount > 50) then
51          if (threadsCount < 10) then
```

PUC-Rio - Certificação Digital Nº 1212362/CA

```
52              -- Creating one thread
53              leda.pool.add(pool,1)
54          end
55      elseif (threadsCount > 2) then
56          -- Removing one thread
57          leda.pool.kill(pool,1)
58      end
59    end
60 end
61
62 -- Fires "onTimer" function each 2 seconds
63 leda.monitoring.addTimer (onTimer,2)
```

Entre as linhas 4 e 10, criamos 3 novos estágios. Entre as linhas 13 e 15, criamos 10 instâncias para cada estágio. Isto quer dizer que cada estágio pode ter no máximo 10 eventos processados em paralelo. Entre as linhas 18 e 20, configuramos a capacidade da fila de entrada de cada estágio com o limite de 100 eventos acumulados. Quando a fila de entrada do estágio ultrapassar este limite, novos eventos serão descartados até que existam menos de 100 eventos acumulados. Entre as linhas 24 e 34, criamos um novo *pool* com 2 *threads* para cada estágio. Até então, os trechos de código inicializam o sistema e são executados apenas uma vez.

O método *onTimer* representa uma *callback* que será chamada periodicamente. Entre as linhas 41 e 44, calculamos o total de eventos acumulados no momento para cada estágio. Para calcular o total de eventos acumulados em cada estágio capturamos o tamanho da fila de eventos, o total de instâncias livres, ou seja, o total de instâncias que ainda não foram combinadas com os eventos e o total de instâncias criadas para cada estágio. Na linha 48, capturamos o número total de *threads* associadas ao *pool* do estágio atual. Entre as linhas 50 e 54, criamos uma nova *thread* caso existam mais de 50 eventos acumulados para o determinado estágio e o *pool* do estágio possua menos de 10 *threads*. Entre as linhas 55 e 58, removemos uma *thread* do *pool* caso exista mais de 2 *threads* associada ao *pool* do estágio. Na linha 63, configuramos o método *onTimer* para ser disparado a cada 2 segundos.

## Implementação

A criação de instâncias, a configuração do número máximo de eventos a serem acumulados em cada fila de eventos, a criação de *pools*, a criação, a remoção e a captura de *threads* já existiam na biblioteca LEDA. Estes métodos foram incluídos em suas respectivas classes de acordo com o modelo

PUC-Rio - Certificação Digital Nº 1212362/CA

de interface desenvolvido neste trabalho. Na criação de *pools* de *threads*, cada *pool* tem uma fila de processamento global. Cada vez que um *pool* é associado a um estágio, este estágio passa a utilizar a fila de processamento global deste *pool*. Os métodos de captura da quantidade de instâncias e de captura do número total de eventos acumulados em um estágio foram criados alterando a estrutura dos estágios para comportar tais contadores. O contador de cada métrica é atualizado quando o número de instâncias ou eventos sofre alteração. Cada contador é atualizado com o uso de *locks* para garantir sua consistência. Para implementar o método de disparo periódico de funções, foi preciso criar uma estrutura de disparo de eventos persistentes. Para armazenar cada uma das funções que devem ser disparadas em um determinado intervalo de tempo configurado através da função *addTimer*, uma lista interna foi criada com todas as funções. Quando um evento de tempo precisa ser disparado, a função correspondente é chamada.

## 4.3
## Cohort

A política *Cohort* já faz uso de um mesmo *pool* de *threads* para todos os estágios. Esta política inicia tratando eventos do primeiro estágio. Quando a fila de eventos do primeiro estágio não possui eventos para processar, as *threads* são alocadas no estágio seguinte e repetem o mesmo procedimento até o fim do pipeline. Ao chegar no último estágio do pipeline, as *threads* são alocadas no estágio anterior até chegarem no primeiro estágio. Ao chegarem novamente no primeiro estágio, os passos se repetem. Para construir um controlador do tipo *Cohort*, precisamos disponibilizar os seguintes métodos em nossa interface:

1. Permitir a criação e remoção de novas *threads* para balancear os recursos da aplicação.

2. Permitir que cada estágio tenha sua fila de processamento, ao invés de compartilhar uma fila global para respeitar a ordem de visitação dos estágios.

3. Permitir configurar a ordem de visitação dos estágios para respeitar a ordem criada na política de *Cohort*.

Para a construção deste controlador, permitimos que o desenvolvedor utilize uma fila de processamento para cada estágio ao invés de utilizar a fila de processamento global. Cada elemento da fila de processamento de um estágio consiste em um evento combinado com uma instância, assim como a fila de processamento global. As *threads* então processam os eventos das filas de cada

PUC-Rio - Certificação Digital Nº 1212362/CA

estágio. Para que a aplicação funcione através da fila de processamento de cada estágio, devemos utilizar o método *usePrivateQueues*. Este método só pode ser chamado uma única vez e faz parte da inicialização do sistema. Após a inicialização, o método não estará mais disponível e sua chamada não fará mais efeito na aplicação. Caso o método *usePrivateQueues* não seja utilizado, LEDA fará uso da fila global. Além das filas de processamento, construímos um método para tornar possível a configuração da ordem de visitação das *threads* chamado de *visitOrder*. Para permitir a configuração da ordem em que as *threads* visitarão os estágios, é possível criar uma tabela de estágios, onde cada posição da tabela representa um estágio. O trecho de código abaixo demonstra um exemplo deste controlador:

```
1 local leda = require 'leda'
2
3 -- Configuring LEDA to use private queues per stage
4 leda.usePrivateQueues()
5
6 -- Creating 2 threads in the global pool
7 leda.pool.add(2)
8
9 -- Creating stages
10 local stage1=leda.stage(
11    function()
12       print("Stage 1 is processing an event...")
13    end)
14
15 local stage2=...
16 local stage3=...
17 local stage4=...
18
19 -- Creating 2 instances for each stage
20 leda.instances.addInstances(stage1,2)
21 leda.instances.addInstances(stage2,2)
22 leda.instances.addInstances(stage3,2)
23 leda.instances.addInstances(stage4,2)
24
25 -- Creating Cohort order
26 local stages = {stage1, stage2, stage3, stage4}
27 local cohortOrder = cohort(stages)
28
29 -- Configuring cohort visit order
30 leda.scheduler.visitOrder(cohortOrder)
```

Na linha 4, configuramos o uso de uma fila de processamento por estágio. Na linha 7, criamos 2 *threads* no *pool* global da aplicação. Entre as linhas 20

PUC-Rio - Certificação Digital Nº 1212362/CA

e 23, criamos 2 instâncias para cada estágio. Na linha 27, utilizamos uma função auxiliar para construir uma tabela de visitação de estágios simulando o caminhamento utilizado na política *Cohort*, visitando os estágios do início para o fim do pipeline e do fim para o início do pipeline. A tabela resultante para o exemplo acima seria *stage1, stage2, stage3, stage4, stage3, stage2*. Na linha 30, a função *visitOrder* recebe a tabela de estágios que indica a ordem em que as *threads* devem visitar cada estágio. Quando uma *thread* termina de processar os eventos do último estágio da tabela, a *thread* retorna para o primeiro estágio, ou seja, para a primeira posição da tabela.

### Implementação

A criação de *threads* e a criação de instâncias já foram explicadas na seção anterior. Para construir os métodos *usePrivateQueues* e *visitOrder* foi necessário modificar a maneira com que as *threads* processam eventos. Quando o método *usePrivateQueues* é chamado, uma fila de processamento é criada para cada estágio. Mesmo que os estágios estejam compartilhando o mesmo *pool* de *threads*, cada estágio passa a utilizar sua fila de processamento. A configuração da ordem de visitação é feita através do método *visitOrder*. Quando o método *visitOrder* é chamado, criamos uma lista encadeada de estágios. Cada elemento dessa lista é uma estrutura com um ponteiro para o estágio atual e um ponteiro para o próximo estágio. Esta lista estipula a ordem em que as *threads* irão processar os eventos dos estágios. Após a configuração, todas as *threads* processam eventos do primeiro estágio da lista. Quando uma *thread* não consegue capturar um evento para processar, a *thread* busca eventos no próximo estágio da lista. Após o processamento dos eventos do último estágio, a *thread* retorna novamente para o início da lista.

PUC-Rio - Certificação Digital Nº 1212362/CA

## 4.4 LRB

A política LRB faz uso de um mesmo *pool* de *threads* para todos os estágios. Esta política prioriza os eventos dos estágios de acordo com sua demanda. Inicialmente, a ordem de visitação das *threads* segue a ordem do *pipeline* da aplicação. Quando a primeira *thread* chega no último estágio do *pipeline*, calculamos a demanda de cada estágio. A demanda de cada estágio é representada pelo número total de eventos que chegaram na fila de eventos do estágio desde a última captura. Com a demanda de cada estágio, calculamos o número total de visitas de cada estágio de acordo com o tamanho do *array* que define a ordem de visitação das *threads*, onde estágios com maior demanda

recebem um maior número de visitas conforme explicado na seção 3.3. Para construir um controlador do tipo *LRB*, precisamos disponibilizar os seguintes métodos em nossa interface:

1. Permitir a criação e remoção de novas *threads* para balancear os recursos da aplicação.
2. Permitir que cada estágio tenha sua fila de processamento, ao invés de compartilhar uma fila global para respeitar a ordem de visitação dos estágios.
3. Permitir configurar a ordem de visitação dos estágios para respeitar a ordem criada na política *LRB*.
4. Permitir a captura da demanda de cada estágio para que seja possível criar uma ordem de visitação de acordo com a demanda de cada estágio.
5. Permitir a criação de *callbacks* que sejam disparadas após uma das *threads* visitar o último estágio da ordenação para que possamos recalcular a ordem de visitação das *threads*.

O trecho de código abaixo demonstra um exemplo deste controlador:

```
local leda = require 'leda'

-- Configuring LEDA to use private queues per stage
leda.usePrivateQueues()

-- Creating 2 threads in the global pool
lstage.pool.add(2)

-- Creating stages
local stage1=lstage.stage(
   function()
      print("Stage 1 is processing an event...")
   end)

local stage2=...
local stage3=...

-- Creating 2 instances for each stage
leda.instances.addInstances(stage1,2)
leda.instances.addInstances(stage2,2)
leda.instances.addInstances(stage3,2)

```

PUC-Rio - Certificação Digital Nº 1212362/CA

```
23 local stages = {stage1, stage2, stage3}
24
25 -- Configuring new visit order
26 leda.scheduler.visitOrder(stages)
27
28 -- Sorting stages table by number of visits
29 function compare(a,b)
30   return a.visits > b.visits
31 end
32
33 -- Callback fired when a thread visits
34 -- the last stage in the visits order array
35 lastStageWasFocused = function ()
36    local visitOrder = {}
37    local total = 0
38    local inputCount = 0
39
40    -- Capture input count for each stage
41    for i,stage in ipairs(stages) do
42       inputCount = leda.monitoring.getInputCount(stage)
43       total      = total + inputCount
44
45       table.insert(visitOrder, { stage = stage, load = inputCount,
            visits = 0 })
46       leda.monitoring.resetStatistics(stage)
47    end
48
49    local loadRate = 0
50    local arraySize = #stages * 3
51
52    -- Calculating number of visits for each stage
53    for i,stage in ipairs(visitOrder) do
54       loadRate = (stage.load * 100) / total
55       stage.visits = math.ceil((loadRate * arraySize) / 100)
56    end
57
58    -- Sorting by number of visits in ascending order
59    table.sort(visitOrder, compare)
60
61    -- Build new visit order table according to LRB policy
62    visitOrder = LRB(visitOrder)
63
64    -- Configuring new visit order
65    leda.scheduler.visitOrder(visitOrder)
66 end
67
```

PUC-Rio - Certificação Digital Nº 1212362/CA

```
68 -- Firing "lastStageWasFocused" function when the last stage is
   focused by one thread
69 leda.monitoring.fireLastFocused(lastStageWasFocused)
```

Na linha 4, configuramos o uso de uma fila de processamento por estágio. Na linha 7, criamos 2 *threads* no *pool* global da aplicação. Entre as linhas 19 e 21, criamos 2 instâncias para cada estágio. Na linha 26, criamos a ordem de visitação das *threads* como a ordem do *pipeline*. Entre as linhas 29 e 31, utilizamos o método *compare* para ordenar os estágios de acordo com o número de visitas de forma ascendente. Até então, os trechos de código inicializam o sistema e são executados apenas uma vez.

O método *lastStageWasFocused* representa uma *callback* que será chamada toda vez que a primeira thread chegar no último estágio da atual ordenação. Na linha 42, a demanda de cada estágio é capturada. Em nosso trabalho, calculamos a demanda de cada estágio capturando o número total de eventos que chegaram para o estágio desde a última medição. Na linha 46, reiniciamos as estatísticas guardadas para cada estágio. Entre as linhas 53 e 54, calculamos o número total de vezes que cada estágio deve aparecer na tabela que representa a ordem de visitação das *threads*. Na linha 59 ordenamos a tabela de estágios em relação ao número de vezes que cada estágio deverá aparecer chamando a função auxiliar *compare*, definida nas linhas 29 a 31. Na linha 62, utilizamos uma função auxiliar para construir uma tabela de visitação de estágios simulando o caminhamento utilizado na política LRB, visitando mais vezes os estágios que tiverem maior demanda, conforme explicado na seção 3.3. Na linha 65, uma nova ordem de visitação é configurada. Na linha 69, o método *lastStageWasFocused* é registrado como uma *callback* a ser disparada quando a primeira *thread* visitar o último estágio da ordenação.

PUC-Rio - Certificação Digital Nº 1212362/CA

### Implementação

A criação de *threads*, a criação de instâncias e a implementação dos métodos *usePrivateQueues* e *visitOrder* já foram apresentadas na seção anterior. Para a construção do método *fireLastFocused*, criamos um contador de visitas com valor inicial 0 para cada estágio. Cada vez que uma *thread* visita um estágio, o contador de visitas do estágio é incrementado com o uso de um *lock*. Quando este contador assume o valor 1, a função configurada como parâmetro no método *fireLastFocused* é disparada. Quando o contador de visitas atinge o número de *threads* do aglomerado, ou seja, quando o estágio foi visitado por todas as *threads* do aglomerado, o contador de visitas recebe novamente o valor 0.

Para a construção do método *getInputCount*, construímos um contador de novos eventos dentro da estrutura de cada estágio. Esse contador é incrementado a cada novo evento que chega a um estágio. O método *resetStatistics* atribui o valor 0 para todas as métricas armazenadas pelo estágio passado como parâmetro. Na implementação das métricas de monitoramento por estágio, a interface que criamos em Leda reúne métodos básicos, como por exemplo a captura de eventos processados. Expor métodos de monitoramento básicos permite com que desenvolvedores criem novas métricas, como por exemplo taxa de serviço e demanda de cada estágio.

## 4.5 DBR

A política DBR (Gordon, 2010) prioriza os eventos dos estágios com a menor taxa de serviço. A cada estágio visitado com sucesso, ou seja, com pelo menos um evento processado, as *threads* retornam ao primeiro estágio da ordenação. Quando as *threads* visitam o último estágio, a priorização dos estágios é refeita de acordo com sua taxa de serviço. Como todos os estágios já receberam visitas das *threads*, a taxa de serviço de cada estágio pode ter sofrido alteração. Para construir este controlador precisamos disponibilizar os seguintes métodos na interface de LEDA:

1. Permitir a criação e remoção de novas *threads* para balancear os recursos da aplicação.
2. Permitir que cada estágio tenha sua fila de processamento, ao invés de compartilhar uma fila global para respeitar a ordem de visitação dos estágios.
3. Permitir configurar a ordem de visitação dos estágios para que estágios possam ser ordenados de acordo com taxa de serviço.
4. Permitir que as *threads* reiniciem do primeiro estágio da ordenação a cada visita bem sucedida para simular o caminhamento da política DBR.
5. Permitir a criação de *callbacks* que sejam disparadas após uma das *threads* visitar o último estágio da ordenação para que possamos recalcular a ordem de visitação das *threads*.
6. Permitir que a taxa de serviço de cada estágio possa ser capturada para que possamos ordená-los e configurar a ordem de visitação das *threads*.

O trecho de código abaixo demonstra um exemplo deste controlador:

PUC-Rio - Certificação Digital Nº 1212362/CA

```lua
local leda = require 'leda'

-- Configuring LEDA to use private queues per stage
leda.usePrivateQueues()

-- Creating 2 threads in the global pool
lstage.pool.add(2)

-- Creating stages
local stage1=lstage.stage(
   function()
      print("Stage 1 is processing an event...")
   end)

local stage2=...
local stage3=...

-- Creating 2 instances for each stage
leda.instances.addInstances(stage1,2)
leda.instances.addInstances(stage2,2)
leda.instances.addInstances(stage3,2)

local stages = {stage1, stage2, stage3}

-- Configuring new visit order
leda.scheduler.visitOrder(stages)

-- Restarting at first stage in each successful threads visit
leda.scheduler.restartAtIndex(1)

-- Sorting polling table by service rate
function compare(a,b)
  local aRate = leda.monitoring.getInputCount(a)
  local bRate = leda.monitoring.getInputCount(b)

  if (aRate ~= 0) then
     aRate = (leda.monitoring.getProcessedCount(a) * 100) / aRate
  end

  if (bRate ~= 0) then
    bRate = (leda.monitoring.getProcessedCount(b) * 100) / bRate
  end

  return aRate < bRate
end

lastStageWasFocused = function ()
```

PUC-Rio - Certificação Digital Nº 1212362/CA

```
48    -- Sorting by rate
49    table.sort(stages, compare)
50
51    -- Reseting statistics for all stages
52    for i,stage in ipairs(stages) do
53       leda.monitoring.resetStatistics(stage)
54    end
55
56    -- Configuring new visit order
57    leda.scheduler.visitOrder (stages)
58 end
59
60 -- Firing "lastStageWasFocused" function when the last stage is
      focused by one thread
61 leda.monitoring.fireLastFocused(lastStageWasFocused)
```

Na linha 4, configuramos o uso de uma fila de processamento por estágio. Na linha 7, criamos 2 *threads* no *pool* global da aplicação. Entre as linhas 19 e 21, criamos 2 instâncias para cada estágio. Na linha 26, configuramos a ordem de visitação inicial. Na linha 29, configuramos as *threads* para sempre retornarem ao primeiro estágio da ordenação a cada visita feita com sucesso. Até então, os trechos de código inicializam o sistema e são executados apenas uma vez. O método *lastStageWasFocused* representa uma *callback* que será chamada toda vez que a primeira thread chegar no último estágio da atual ordenação. Na linha 49 ordenamos os estágios de acordo com a taxa de processamento de cada um utilizando o método *compare*. Calculamos a taxa de processamento de cada estágio calculando o percentual entre o número total de eventos que chegaram na fila de eventos de um estágio e o número total de eventos processados por este estágio. Na linha 53 reiniciamos as estatísticas de cada estágio. Na linha 57 configuramos a nova ordem de visitação das *threads*. Na linha 61 configuramos o disparo do método *lastStageWasFocused* quando a primeira *thread* visitar o último estágio da ordenação.

PUC-Rio - Certificação Digital Nº 1212362/CA

### Implementação

A criação de *threads*, a criação de instâncias e a implementação dos métodos *usePrivateQueues*, *visitOrder*, *getInputCount*, *fireLastFocused* e *resetStatistics* já foram apresentadas nas seções anteriores. Para a construção do método *restartAtIndex*, criamos uma variável booleana que indica se a *thread* conseguiu ou não processar algum evento do estágio que está em foco. Cada *thread* possui sua variável e esta variável não é compartilhada entre as *threads*, não precisando de nenhum mecanismo de trava para atualizá-la. Antes de bus-

car pelo próximo estágio da lista encadeada, esta variável é verificada. Caso algum evento tenha sido processado no atual estágio, a *thread* retorna ao primeiro estágio da lista. Caso contrário, o próximo estágio da lista é capturado pela *thread*. Para a construção do método *getProcessedCount*, incluímos um contador de eventos processados na estrutura de cada estágio. Cada vez que um evento do estágio é processado, o contador é incrementado com o uso de um *lock*.

## 4.6 SRPT

O controlador SRPT (Harchol-Balter et al., 2001) trata as requisições de trás para frente, priorizando os eventos dos estágios que estão no fim do pipeline. As *threads* retornam novamente para o último estágio do pipeline cada vez que visitam um estágio com sucesso, ou seja, com pelo menos um evento processado. Para implementar este controlador, precisamos disponibilizar os seguintes métodos na interface de LEDA:

1. Permitir a criação e remoção de novas *threads* para balancear os recursos da aplicação.
2. Permitir que estágios possam ser priorizados para criar a ordem de processamento através da fila de processamento global.
3. Permitir que cada estágio tenha sua fila de processamento, ao invés de compartilhar uma fila global para respeitar a ordem de visitação dos estágios.
4. Permitir configurar a ordem de visitação dos estágios para que estágios possam ser ordenados de acordo com taxa de serviço.
5. Permitir que as *threads* reiniciem do primeiro estágio da ordenação a cada visita bem sucedida para simular o caminhamento da política SRPT.

Através da interface disponibilizada pelo LEDA, existem duas maneiras diferentes de implementar esta política. A primeira implementação é através da fila global: como as prioridades dos estágios nunca mudam, é possível utilizar a fila global para priorizar eventos de estágios no fim do pipeline. Quando um novo evento for inserido na fila global, ele será posicionado na fila de acordo com sua prioridade configurada. Desta maneira, o funcionamento do controlador será semelhante ao de retornar ao estágio final, já que os eventos dos estágios finais da aplicação serão alocados no início da fila global, e por consequência, processados primeiro. O trecho de código abaixo demonstra um

PUC-Rio - Certificação Digital Nº 1212362/CA

exemplo de controlador do tipo *SRPT* utilizando a fila global:

```
1 local leda = require 'leda'
2
3 -- Creating 2 threads in the global pool
4 lstage.pool.add(2)
5
6 -- Creating stages
7 local stage1=leda.stage(
8    function()
9       print("Stage 1 is processing an event...")
10   end)
11
12 local stage2=...
13 local stage3=...
14
15 -- Creating 2 instances for each stage
16 leda.instances.addInstances(stage1,2)
17 leda.instances.addInstances(stage2,2)
18 leda.instances.addInstances(stage3,2)
19
20 -- Prioritizing final stages
21 leda.scheduler.setPriority(stage3,3)
22 leda.scheduler.setPriority(stage2,2)
23 leda.scheduler.setPriority(stage1,1)
```

Na linha 4, criamos 2 *threads* no *pool* global da aplicação. Entre as linhas 16 e 18, criamos 2 instâncias para cada estágio. Entre as linhas 21 e 23 configuramos a prioridade de cada estágio. Estágios com maior prioridade terão seus eventos alocados no início da fila global.

A segunda implementação é através da fila de processamento de cada estágio: uma ordem de visitação é configurada e as *threads* retornam ao último estágio do pipeline a cada estágio visitado com sucesso. Isto é possível através do uso dos métodos *usePrivateQueues*, *visitOrder* e *restartAtIndex* já explicados nas seções anteriores. O trecho de código abaixo demonstra um exemplo de controlador do tipo *SRPT* utilizando a fila de processamento de cada estágio:

```
1 local leda = require 'leda'
2
3 -- Configuring LEDA to use private queues per stage
4 leda.usePrivateQueues()
5
6 -- Creating 2 threads in the global pool
```

PUC-Rio - Certificação Digital Nº 1212362/CA

```
7 lstage.pool.add(2)
8
9 -- Creating stages
10 local stage1=leda.stage(
11    function()
12       print("Stage 1 is processing an event...")
13    end)
14
15 local stage2=...
16 local stage3=...
17
18 -- Creating 2 instances for each stage
19 leda.instances.addInstances(stage1,2)
20 leda.instances.addInstances(stage2,2)
21 leda.instances.addInstances(stage3,2)
22
23 local stages = {stage3, stage2, stage1}
24
25 -- Configuring new visit order
26 leda.scheduler.visitOrder(stages)
27
28 -- Restarting at the last stage at each successful visit
29 leda.scheduler.restartAtIndex(1)
```

Na linha 4, configuramos o uso de uma fila de processamento por estágio. Na linha 7, criamos 2 *threads* no *pool* do aglomerado. Entre as linhas 19 e 21, criamos 2 instâncias para cada estágio. Na linha 26, configuramos a ordem de visitação das *threads* iniciando o processo pelo estágio final. Na linha 29, configuramos as *threads* para sempre retornarem ao *stage3* a cada visita feita com sucesso.

As duas implementações podem apresentar diferentes resultados de desempenho por conta das diferentes formas de visitação feita pelas *threads* da aplicação. Enquanto as *threads* da implementação através da fila de processamento global processam eventos apenas buscando em uma fila, na implementação por filas privadas as *threads* processam eventos visitando cada estágio da aplicação.

### Implementação

A criação de *threads*, a criação de instâncias e a implementação já foram apresentadas nas seções anteriores. Na implementação pela fila de processamento global, utilizamos o método *setPriority* que já existia na

PUC-Rio - Certificação Digital Nº 1212362/CA

biblioteca LEDA. Este método foi incluído em sua classe de acordo com o modelo de interface desenvolvido neste trabalho. Na implementação pela fila de processamento por estágio, utilizamos os método *usePrivateQueues*, *visitOrder* e *restartAtIndex* que já foram explicados nas seções anteriores.

## 4.7 Fila única de eventos

A biblioteca LEDA funciona nativamente com uma fila única de processamento, não precisando de alterações para a construção de controladores deste tipo. Por padrão, todos os estágios possuem a mesma prioridade e seus eventos são processados em ordem de chegada conforme explicado na seção 3.6.

## 4.8 Workstealing

Para implementar técnicas do tipo *Workstealing* com a interpretação discutida na seção 3.7, precisamos de métodos que nos permitam o roubo de recursos, ou seja, métodos que redirecionem *threads* de um *pool* para outro. Esta aplicação apenas faz sentido dentro das arquiteturas de estágios se pensarmos em um modelo de *pool* de *threads* por estágio. Caso exista algum estágio com poucos recursos, é possível roubar *threads* de um outro estágio. Para construir este controlador precisamos disponibilizar os seguintes métodos na interface de LEDA:

1. Permitir a criação de *pools* de *threads* associados a estágios individuais.
2. Permitir a criação e remoção de novas *threads* para balancear os recursos da aplicação.
3. Permitir que a taxa de serviço de cada estágio possa ser capturada para que possamos direcionar *threads* para *pools* de estágios com menor taxa de serviço.
4. Permitir o roubo de *threads* entre os estágios.

Nesta implementação de *Workstealing*, estágios são ordenados de acordo com sua taxa de serviço de forma ascendente. Estágios com a menor taxa de serviço roubam *threads* de estágios com maior taxa de serviço, ou seja, estágios iniciais da ordenação roubam *threads* dos estágios finais da ordenação. Para evitar muitas realocações de *threads* entre *pools*, limitamos o roubo de *threads* a 2 estágios, onde os 2 estágios com menor taxa de serviço podem roubar *threads* dos 2 estágios com maior taxa de serviço.

PUC-Rio - Certificação Digital Nº 1212362/CA

Cada estágio possui uma fila de roubo. Cada evento na fila de roubo é constituído de um ponteiro para o *pool* que requer novas *threads* e um contador, que indica a quantidade de *threads* demandada pelo *pool*. Quando algum evento de roubo é criado na fila de roubo de algum estágio, significa que existe algum estágio demandando por *threads* deste estágio.

Cada vez que uma *thread* de um *pool* fica ociosa, esta *thread* verifica na fila de roubo se existe algum *pool* demandando *threads*. Caso exista, esta *thread* passa a apontar para o *pool* do estágio que requisitou mais *threads*. Caso mais de uma *thread* tenha sido requisitada pelo mesmo estágio, o contador é atualizado com seu valor atual menos 1 e colocado no fim da fila de roubo novamente. Colocar eventos que requerem mais de uma *thread* no fim da fila após o primeiro roubo permite que o roubo de *threads* seja balanceado entre os estágios. O trecho de código abaixo demonstra um exemplo de controlador do tipo *Workstealing*:

```
1  local leda = require 'leda'
2
3  -- Creating stages
4  local stage1=leda.stage(
5     function()
6        print("Stage 1 is processing an event...")
7     end)
8
9  local stage2=...
10 local stage3=...
11
12 -- Creating 10 instances for each stage
13 leda.instances.addInstances(stage1,10)
14 leda.instances.addInstances(stage2,10)
15 leda.instances.addInstances(stage3,10)
16
17 -- Creating new thread pool with 2 threads
18 -- and associating with each stage
19 local pool1 = leda.pool.new()
20 leda.pool.add(pool1,2)
21 leda.pool.setPool(stage1,pool1)
22
23 local pool2 = leda.pool.new()
24 leda.pool.add(pool2,2)
25 leda.pool.setPool(stage2,pool2)
26
27 local pool3 = leda.pool.new()
28 leda.pool.add(pool3,2)
29 leda.pool.setPool(stage3,pool3)
```

PUC-Rio - Certificação Digital Nº 1212362/CA

```
30
31 local stages = {stage1,stage2,stage3}
32
33 function compare(a,b)
34   local aRate = leda.monitoring.getInputCount(a)
35   local bRate = leda.monitoring.getInputCount(b)
36
37   if (aRate ~= 0) then
38      aRate = (leda.monitoring.getProcessedCount(a) * 100) / aRate
39   end
40
41   if (bRate ~= 0) then
42     bRate = (leda.monitoring.getProcessedCount(b) * 100) / bRate
43   end
44
45   return aRate < bRate
46 end
47
48 onTimer = function ()
49    -- Sort stages by service rate in descending order
50    table.sort(stages,compare)
51
52    local first = 1
53    local last  = #stages
54
55    for i=1,2,1 do
56       -- Capturing each stage pool
57       local firstPool = leda.pool.getPool (stages[first])
58       local lastPool = leda.pool.getPool (stages[last])
59
60       -- Capturing threads count for each pool
61       local lastPoolSize = leda.pool.getThreadsCount(lastPool)
62       local firstPoolSize = leda.pool.getThreadsCount(firstPool)
63
64       -- Capturing instances count for the first stage to check if
65       -- it can get one more thread
66       local instancesCount = leda.instances.getInstancesCount(
            stages[first])
67
68       -- Stealing one thread
69       if (lastPoolSize > 1 and instancesCount >= (firstPoolSize +
            1)) then
70          leda.scheduler.steal(stages[first],stages[last],1)
71       end
72
73       first = first + 1
74       last  = last - 1
```

PUC-Rio - Certificação Digital Nº 1212362/CA

```
75     end
76
77     -- Reseting statistics for all stages
78     for i,stage in ipairs(stages) do
79         leda.monitoring.resetStatistics(stage)
80     end
81 end
82
83 -- Firing "onTimer" function every 10 seconds
84 leda.monitoring.addTimer (onTimer,10)
```

Entre as linhas 19 e 29, criamos um *pool* com 2 *threads* para cada estágio. Na linha 48, o método *onTimer* representa uma *callback* que será chamada periodicamente. Na linha 50, os estágios são ordenados de acordo com sua demanda de forma ascendente. Entre as linhas 57 e 62, capturamos o número total de *threads* do estágio inicial e do estágio final de acordo com o índice atual. Na linha 70, um estágio com menor taxa de serviço rouba uma *thread* de um estágio com maior taxa de serviço caso ele tenha o número de instâncias concorrentes suficientes para poder receber mais uma *thread* em seu *pool*. Na linha 79, reiniciamos as estatísticas de todos os estágios. Na linha 84, o método *onTimer* é configurado para disparar a cada 10 segundos.

PUC-Rio - Certificação Digital Nº 1212362/CA

### Implementação

A criação de *threads*, a criação de novos *pools*, a criação de instâncias e a implementação dos métodos *getInputCount*, *getProcessedCount*, *getThreadsCount*, *resetStatistics*, *addTimer* e *getPool* já foram apresentadas nas seções anteriores. Para a criação do método *steal*, construímos uma nova fila para cada *pool* de *threads*. Cada fila foi chamada de fila de roubo. Cada elemento da fila de roubo consiste em um *struct* com um ponteiro para o *pool* que está requerendo novas *threads* e um contador, indicando o número de *threads* que o *pool* está demandando. Uma *thread* verifica se existe algum elemento na fila de roubo de seu *pool* cada vez que termina de processar um evento. Caso exista algum *pool* demandando novas *threads*, a *thread* que terminou de processar um evento passa a apontar para o *pool* que demandou novas *threads*. O contador de número de *threads* a serem direcionadas é então decrementado e este *struct* é passado para o fim da fila de roubo. Colocar *structs* da fila de roubo no fim da fila de roubo após algum redirecionamento de *threads* faz com que o roubo de recursos entre *pools* seja balanceado. Caso inúmeros *pools* demandem novas *threads* de um mesmo *pool*, os *pools* receberão o número de *threads* na mesma proporção.

## 4.9 Discussão

Neste capítulo vimos como é possível implementar cada uma das políticas descritas no capítulo anterior através da interface disponibilizada por LEDA. Alguns métodos já existiam na biblioteca e diversos outros precisaram ser construídos dentro da interface. O uso da interface permite facilitar a construção de novos controladores bem como permite que os desenvolvedores combinem o comportamento destas técnicas. Para o monitoramento das aplicações, a interface permite a construção de novas estatísticas a partir de métodos mais básicos, como por exemplo calcular a taxa de serviço através dos eventos que chegaram na fila de eventos de um estágio e dos eventos processados de um estágio. Com este trabalho emergiram algumas semelhanças e diferenças entre políticas, possibilitando com que métodos fossem compartilhados entre controladores. Através do modelo de interface, estas semelhanças e diferenças pôde nos ajudar no entendimento do comportamento dos controladores.

PUC-Rio - Certificação Digital Nº 1212362/CA

# 5
# Aplicações

O objetivo deste capítulo é demonstrar que aplicações diferentes podem reagir de forma distinta com o uso de diferentes controladores. Para isso, desenvolvemos duas aplicações, um tratador de imagens e um servidor HTTP.

Todos os testes foram feitos com uma máquina virtual da AWS (do inglês *Amazon Web Services*) rodando com o sistema operacional Ubuntu 14.04 com 8 CPUs, 15GB de memória, 2.8 GHz e um processador Intel Xeon E5.

## 5.1
## Tratador de imagens

A aplicação de tratamento de imagens foi criada para tornar possível a binarização de Captchas (do inglês *Completely Automated Public Turing test to tell Computers and Humans Apart*) como um processo inicial no reconhecimento de caracteres. Captchas são utilizados em diversos serviços da Internet como uma forma de evitar acessos robotizados. Exemplos de imagens de entrada são mostrados na figura 5.1.

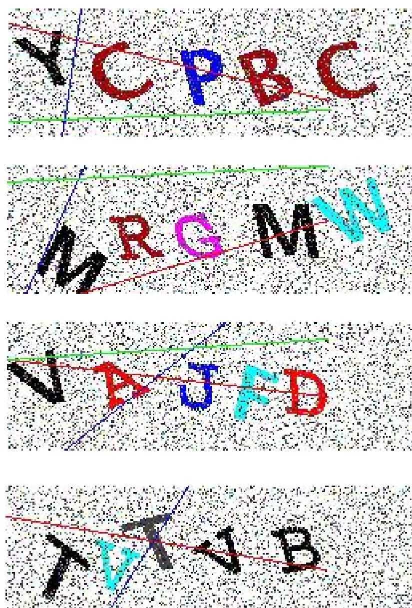

Figura 5.1: Exemplo de Captchas utilizados para a binarização

PUC-Rio - Certificação Digital Nº 1212362/CA

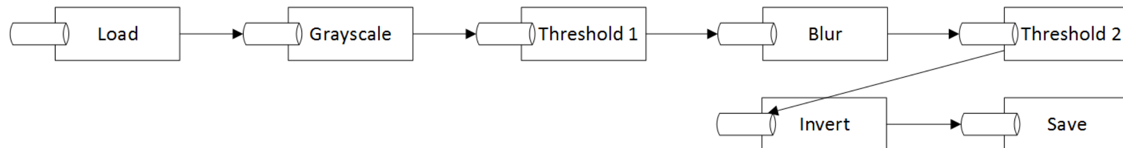


Figura 5.2: Estágios do tratador de imagens

Para a construção desta aplicação utilizamos Lua e C. Para ler e salvar as imagens em disco, utilizamos a biblioteca *Imlib2* (Imlib, 2013). Para aplicar *threshold* nas imagens, utilizamos a biblioteca *OpenCv* (OpenCv, 2011). Desenvolvemos os métodos *blur*, *invert* e *grayscale* em C. Nesta aplicação, os estágios que realizam tratamento de imagem possuem processamento computacional semelhante. A aplicação está dividida em sete estágios, conforme ilustrado na figura 5.2. A descrição de cada estágio é feita abaixo:

1. *Load*
   Este estágio é o responsável por ler as imagens do disco utilizando a biblioteca *Imlib2* e passar o ponteiro de cada imagem para o estágio seguinte.

2. *Grayscale*
   Este estágio é o responsável por transformar as imagens em escalas de cinza. Desenvolvemos este método em C.

3. *Threshold*
   Este estágio é o responsável por aplicar um *threshold* na imagem. *Threshold* é geralmente aplicado em imagens que precisam ser binarizadas. Caso o valor de um *pixel* esteja acima de um *threshold*, o *pixel* recebe a cor preta, caso contrário, a cor branca. Utilizamos a biblioteca *OpenCv* para aplicar este tratamento.

4. *Blur*
   Este estágio é o responsável por ofuscar a imagem. Necessário para que o próximo *threshold* possa remover a maioria dos ruídos da imagem. Desenvolvemos este método em C.

5. *Invert*
   Este estágio é o responsável por inverter cores das imagens, trocando a cor branca por preta e vice-versa. Desenvolvemos este método em C.

6. *Save*
   Estágio final da aplicação. Responsável por salvar em disco as imagens utilizando a biblioteca *Imlib2*.

PUC-Rio - Certificação Digital Nº 1212362/CA

A figura 5.3 demonstra o resultado de cada passo utilizado para binarizar cada imagem.

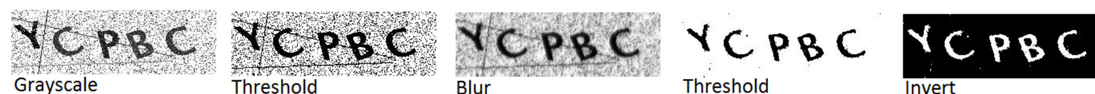


Figura 5.3: Tratamentos aplicados em uma imagem

Na subseção a seguir, descreveremos os testes realizados com diferentes controladores para esta aplicação.

### 5.1.1 Testes

Realizamos os testes com 5.000 imagens que têm dimensões variando entre 256 x 80 e 2.560 x 1.600 *pixels*. A escolha no número de *threads* para cada aplicação se deve ao fato de que precisamos testar as aplicações com um número de *threads* igual ou superior ao número de CPU's na máquina. Cada estágio tem o número de instâncias igual ao número de *threads* de seu *pool*. Como técnicas do tipo SEDA e *Workstealing* utilizam *pools* para estágios individuais, realizamos os testes com 2 *threads* por estágio para evitar manter alguma CPU sem processamento. A tabela 5.1 mostra o tempo total em segundos que cada um dos controladores levou para processar todas as imagens.

| Política | Threads | Tempo (em seg.) |
|---|---:|---:|
| Fila única | 14 | 437,48 |
| SEDA | 28 (4 p/ estágio) | 477,48 |
| SRPT (global) | 14 | 488,64 |
| Cohort | 14 | 584,64 |
| DBR | 14 | 591,44 |
| SEDA | 14 (2 p/ estágio) | 620,04 |
| LRB | 14 | 623,31 |
| Workstealing | 14 (2 p/ estágio) | 627,52 |
| SRPT (privada) | 14 | 740,72 |

Tabela 5.1: Tempo de processamento de 5.000 imagens em segundos por política

Com este resultado, podemos observar que o controlador do tipo fila única foi o que obteve melhor resultado. Mesmo utilizando o controlador do tipo SEDA com um maior número de *threads*, o controlador por fila única obteve melhor resultado em relação aos outros. Aplicações onde todos os estágios possuem demanda e tempo de processamento semelhantes não se beneficiam com técnicas de *Workstealing*.

PUC-Rio - Certificação Digital Nº 1212362/CA

Aplicações com estágios que têm diferentes taxas de serviço, como por exemplo aplicações onde um estágio faz apenas acesso a uma variável de memória e outro estágio faz agrupamentos em um banco de dados podem se beneficiar com técnicas do tipo *Workstealing*. Podemos observar também que a implementação do controlador SRPT pela fila global obteve maior vantagem do que o controlador SRPT contruído através das filas privadas, que obteve o pior resultado para esta aplicação. A diferença entre as duas implementações da técnica SRPT ocurreu pois a implementação por fila privada exige que as *threads* alternem entre diferentes filas de processamento, já que precisam seguir a ordenação configurada e já que cada estágio possui uma fila de processamento. A técnica do tipo LRB não teve um bom resultado pois os estágios possuem demanda semelhantes. No momento da criação da nova ordem de visitação um dos estágios pode ter recebido mais eventos que outros, recebendo mais visitas. Como o desempenho dos estágios é semelhante, as *threads* acabam visitando de forma desnecessária um maior número de vezes o estágio que no momento da construção estava com alguns eventos a mais.

PUC-Rio - Certificação Digital Nº 1212362/CA

## 5.2 Servidor HTTP

Esta aplicação consiste em um servidor HTTP para atender requisições de páginas estáticas e dinâmicas. A aplicação se divide nos estágios descritos na figura 5.4.

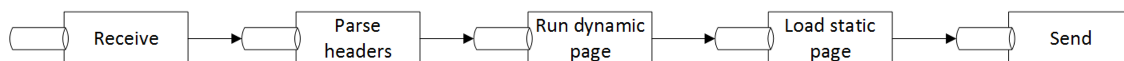


Figura 5.4: Estágios do servidor HTTP

Para a construção desta aplicação utilizamos Lua (Ierusalimschy, 2013). Para tratar requisições utilizamos LuaSocket (Nehab, 2007). A descrição de cada estágio é feita abaixo:

1. *Start*

   Este estágio é o responsável por abrir uma porta de um *socket* e receber novas requisições.

2. *Parse headers*

   Este estágio é o responsável por ler e realizar o parse do cabeçalho da requisição.

3. *Run dynamic page*

   Esta estágio é responsável por processar páginas dinâmicas.

4. *Load static page*
   Este estágio é responsável por ler uma página estática do disco.

5. *Send*
   Este estágio é responsável por combinar o conteúdo estático e dinâmico das páginas e enviar a resposta para o cliente.

Na subseção a seguir, descreveremos os testes realizados com diferentes controladores para esta aplicação.

### 5.2.1 Testes

Para simular requisições, utilizamos a ferramenta Httperf (Mosberger e Jin, 1998). Esta ferramenta foi criada para medir o desempenho de servidores, provendo flexibilidade e facilidade para gerar cargas de trabalho para servidores HTTP. Httperf foi configurado para enviar 30 requisições por segundo até completar 2.000 requisições. A cada segundo, novas requisições são enviadas para o servidor e as conexões anteriores continuam sendo atendidas. Cada requisição envolve uma página estática e dinâmica e todas as requisições são enviadas da mesma máquina.

Para a escolha no número de *threads*, utilizamos a mesma idéia utilizada na aplicação de imagens, não permitindo que qualquer CPU ficasse sem processamento. Cada estágio tem o número de instâncias igual ao número de *threads* de seu *pool*. A tabela 5.2 mostra o tempo de todas as requisições por usuário, ou seja, o tempo total em segundos que um cliente precisou esperar para obter todo o conteúdo de uma requisição:

| Política | Threads | Min. | Máx. | Média | Desvio padrão |
|---|---:|---:|---:|---:|---:|
| SRPT (global) | 10 | 2,03 | 249,02 | 125,56 | 71,79 |
| SEDA | 20 (4 p/ estágio) | 1,48 | 249,65 | 127,87 | 72,50 |
| SEDA | 40 (8 p/ estágio) | 3,31 | 250,28 | 128,86 | 72,53 |
| LRB | 10 | 8,44 | 247,48 | 129,44 | 64,31 |
| DBR | 10 | 9,24 | 249,83 | 129,59 | 71,89 |
| Workstealing | 10 (2 p/ estágio) | 1,32 | 279,78 | 142,48 | 81,23 |
| SEDA | 10 (2 p/ estágio) | 1,23 | 275,44 | 144,60 | 82,61 |
| Fila única | 10 | 2,65 | 249,78 | 171,58 | 45,22 |
| Cohort | 10 | 165,97 | 250,72 | 208,22 | 24,31 |
| SRPT (privado) | 10 | 169,27 | 254,0 | 211,33 | 24,31 |

Tabela 5.2: Tempo de requisições atentidas por usuário

Podemos observar os resultados diferentes em relação à aplicação de imagens. Enquanto a técnica de fila única teve um resultado inferior nesta

PUC-Rio - Certificação Digital Nº 1212362/CA

aplicação, a técnica de SRPT já demonstrou vantagens. A técnica de *Workstealing* obteve um melhor resultado em relação à aplicação de tratamento de imagens, já que nesta aplicação estágios têm diferentes taxas de serviço. Em particular, o estágio que executa páginas dinâmicas tem uma taxa de serviço menor do que todos os demais. Comprovamos a diferença de desempenho entre os estágios medindo periodicamente a taxa de serviço de cada um.

A tabela 5.3 mostra o número de respostas por cliente por segundo para cada controlador. Analisando o valor mínimo e o máximo conseguimos enxergar melhor o comportamento da aplicação com cada um dos controladores. Por exemplo, a aplicação configurada com o controlador do tipo SRPT através das filas privadas ficou um segundo sem responder nenhum cliente porém em outro segundo respondeu 12 requisições. Os controladores LRB, DBR e SRPT com fila global tiveram um resultado 10% melhor do que o controlador SEDA com o mesmo número de *threads* e do que o controlador *Workstealing* em relação a média do número total de respostas por usuário.

| Política | Threads | Min. | Máx. | Média | Desvio padrão |
|---|---|---|---|---|---|
| LRB | 10 | 0.0 | 10.8 | 4.0 | 3.5 |
| DBR | 10 | 0.0 | 9.8 | 4.0 | 3.2 |
| SEDA | 20 (4 p/ estágio) | 2.8 | 6.0 | 4.0 | 0.4 |
| SRPT (global) | 10 | 3.2 | 4.4 | 4.0 | 0.2 |
| Cohort | 10 | 0.0 | 12.0 | 3.9 | 5.6 |
| Fila única | 10 | 0.0 | 9.4 | 3.9 | 3.5 |
| SEDA | 40 (8 p/ estágio) | 1.6 | 5.6 | 3.9 | 0.5 |
| SRPT (privado) | 10 | 0.0 | 12.0 | 3.8 | 5.5 |
| SEDA | 10 (2 p/ estágio) | 2.6 | 6.0 | 3.6 | 0.7 |
| Workstealing | 10 (2 p/ estágio) | 2.6 | 4.6 | 3.6 | 0.3 |

Tabela 5.3: Número de respostas por usuário por segundo

PUC-Rio - Certificação Digital Nº 1212362/CA

## 5.3 Discussão

Neste capítulo analisamos o desempenho de duas diferentes aplicações com o uso de diferentes controladores. Pudemos concluir que aplicações respondem de forma diversa de acordo com cada controlador, constatando a importância da construção de controladores e a vantagem de poder construí-los sem que seja necessário gerar várias versões da mesma aplicação.

A aplicação de tratamento de imagens possui a maioria dos seus estágios com um custo de processamento semelhante. O servidor HTTP possui um número maior de estágios com custo diferente. Por conta dos diferentes custos de processamento de cada aplicação obtivemos diferentes resultados. Os resultados demonstrados neste capítulo foram condensados. Realizamos testes

com outros números de threads e o desempenho das aplicações utilizando cada um dos controladores foi equivalente ao apresentado neste capítulo.

PUC-Rio - Certificação Digital Nº 1212362/CA

# 6
# Conclusão

O objetivo deste trabalho é criar um modelo de interface que permite a construção de novos controladores sem que seja necessário criar inúmeras implementações da mesma aplicação. Através do estudo de diversas técnicas, pudemos analisar os métodos necessários para a construção de cada controlador. Alguns métodos já existiam na biblioteca LEDA e outros precisaram ser construídos e incluídos na interface para que todos os controladores pudessem ser construídos. O modelo de interface permite que as técnicas exploradas possam ser combinadas, permitindo a construção de novos controladores, além de facilitar a escolha da melhor configuração para cada aplicação, já que criamos métodos que permitem capturar estatísticas dos estágios.

Neste trabalho pudemos comprovar que diferentes aplicações podem responder de forma diversa a determinados controladores. Realizamos os testes implementando 2 diferentes aplicações, um tratador de imagens e um servidor HTTP. Estes testes provam a importância destas técnicas e demonstra o quão importante um modelo de interface pode se tornar para aplicações, possibilitando a criação e combinação de técnicas abordadas por cada controlador.

## 6.1
## Contribuições

Ao desenvolver este trabalho, buscamos estudar técnicas de controle e criar uma interface com os métodos necessários para a construção destas técnicas. O modelo de interface permite que diversos controladores possam ser construídos e permite também que técnicas sejam combinadas para gerar novos controladores. Nossas principais contribuições são:

- A investigação dos métodos necessários para a construção de controladores bem como o desenvolvimento de uma interface que permita a contrução de controladores
- A comprovação de que aplicações distintas reagem de forma diversa com o uso de diferentes controladores

PUC-Rio - Certificação Digital Nº 1212362/CA

– Uma adaptação da técnica de *Workstealing* para as arquiteturas baseadas em estágios

## 6.2 Trabalhos futuros

Temos interesse em investigar o desempenho de controladores balanceando o desempenho de aplicações em máquinas distintas, adaptando a interface para facilitar a construção de novos controladores que permitam o monitoramento e o gerenciamento de recursos em diferentes máquinas. Além disso, temos o interesse em criar novos controladores combinando técnicas diferentes.

PUC-Rio - Certificação Digital Nº 1212362/CA

# 7
# Referências Bibliográficas

AMAZON. **What is Amazon Simple Queue Service?** 2006. http://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/Welcome.html. [Online; accessed 28-September-2014].

BEHLENDORF, B. et al. **What is the Apache HTTP Server Project?** 1995. http://httpd.apache.org/ABOUT_APACHE.html. [Online; accessed 25-August-2014].

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling multithreaded computations by work stealing. **J. ACM**, ACM, New York, NY, USA, v. 46, n. 5, p. 720–748, set. 1999. ISSN 0004-5411. Disponível em: <http://doi.acm.org/10.1145/324133.324234>.

BOXMA, O.; LEVY, H.; WESTSTRATE, J. Efficient visit orders for polling systems. **Perform. Eval.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 18, n. 2, p. 103–123, set. 1993. ISSN 0166-5316. Disponível em: <http://dx.doi.org/10.1016/0166-5316(93)90031-O>.

BURTON, F. W.; SLEEP, M. R. Executing functional programs on a virtual tree of processors. In: **Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture**. New York, NY, USA: ACM, 1981. (FPCA '81), p. 187–194. ISBN 0-89791-060-5. Disponível em: <http://doi.acm.org/10.1145/800223.806778>.

CHEN, X.; MOHAPATRA, P.; CHEN, H. An admission control scheme for predictable server response time for web accesses. In: **In Proceedings of the 10th World Wide Web Conference, Hong Kong**. [S.l.: s.n.], 2001. p. 545–554.

DEAN, J.; GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. **Commun. ACM**, ACM, New York, NY, USA, v. 51, n. 1, p. 107–113, jan. 2008. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1327452.1327492>.

PUC-Rio - Certificação Digital Nº 1212362/CA

FOSTER, I. **Designing and building parallel programs : concepts and tools for parallel software engineering**. Reading, Mass: Addison-Wesley, 1995. ISBN 978-0201575941.

FOUNDATION, A. S. **Apache Hadoop**. 2006. http://hadoop.apache.org/docs/current/. [Online; accessed 14-September-2014].

FREEMAN, E.; HUPFER, S.; ARNOLD, K. **JavaSpaces principles, patterns, and practice**. Reading, MA: Addison-Wesley, 1999. ISBN 978-0201309553.

GAUD, F. et al. Efficient workstealing for multicore event-driven systems. In: **Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems**. Washington, DC, USA: IEEE Computer Society, 2010. (ICDCS '10), p. 516–525. ISBN 978-0-7695-4059-7. Disponível em: <http://dx.doi.org/10.1109/ICDCS.2010.55>.

GOLDRATT, E. M. **What is this thing called theory of constraints and how should it be implemented**. Croton-on-Hudson, N.Y: North River Press, 1990. ISBN 978-0884271666.

GONG, M.; WILLIAMSON, C. Quantifying the properties of SRPT scheduling. In: **Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on**. [S.l.: s.n.], 2003. p. 126–135. ISSN 1526-7539.

GORDON, M. E. **Stage Scheduling for CPU-intensive servers**. Tese (Doutorado) — University of Cambridge, 2010.

HAN, B. et al. An improved staged event driven architecture for master-worker network computing. In: **Cyber-Enabled Distributed Computing and Knowledge Discovery, 2009. CyberC '09. International Conference on**. [S.l.: s.n.], 2009. p. 184–190.

HARCHOL-BALTER, M. et al. SRPT scheduling for web servers. In: **Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing**. London, UK, UK: Springer-Verlag, 2001. (JSSPP '01), p. 11–20. ISBN 3-540-42817-8. Disponível em: <http://dl.acm.org/citation.cfm?id=646382.689688>.

HARCHOL-BALTER, M. et al. Size-based scheduling to improve web performance. **ACM Trans. Comput. Syst.**, ACM, New York, NY, USA, v. 21, n. 2, p. 207–233, maio 2003. ISSN 0734-2071. Disponível em: <http://doi.acm.org/10.1145/762483.762486>.

PUC-Rio - Certificação Digital Nº 1212362/CA

HARIZOPOULOS, S.; AILAMAKI, A. Affinity scheduling in staged server architectures. **Technical report, Carnegie Mellon University**, 2002.

IERUSALIMSCHY, R. **Programming in Lua**. Lua.org, 2013. ISBN 859037985X. Disponível em: <http://www.amazon.com/exec/obidos/ASIN/859037985X/lua-pilindex-20>.

IMLIB. **Imlib2 API Reference**. 2013. http://alien.cern.ch/cache/imlib2-1.0.6/doc/. [Online; accessed 03-January-2015].

KANODIA, V.; KNIGHTLY, E. W. Multi-class latency-bounded web services. In: **Quality of Service, 2000. IWQOS. 2000 Eighth International Workshop on**. [S.l.: s.n.], 2000. p. 231–239.

LARUS, J. R.; PARKES, M. Using cohort scheduling to enhance server performance. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 36, n. 8, p. 182–187, ago. 2001. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/384196.384222>.

LEE, E. A. The problem with threads. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 39, n. 5, p. 33–42, maio 2006. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.2006.180>.

LI, Z. et al. Auto-tune design and evaluation on staged event-driven architecture. In: **Proceedings of the 1st Workshop on MOdel Driven Development for Middleware (MODDM '06)**. New York, NY, USA: ACM, 2006. (MODDM '06), p. 1–6. ISBN 1-59593-423-5. Disponível em: <http://doi.acm.org/10.1145/1169086.1169089>.

MICROSOFT. **Optimizing IIS Performance**. 2011. http://msdn.microsoft.com/en-us/library/ee377050.aspx. [Online; accessed 25-August-2014].

MOSBERGER, D.; JIN, T. httperf - a tool for measuring web server performance. **HP Research Labs. Hewlett-Packard Co.**, 1998. Disponível em: <http://www.hpl.hp.com/research/linux/httperf/wisp98/httperf.pdf>.

NEHAB, D. **Network support for the Lua language**. 2007. http://w3.impa.br/~diego/software/luasocket/. [Online; accessed 03-January-2015].

OPENCV. **OpenCv documentation**. 2011. http://docs.opencv.org/index.html. [Online; accessed 03-January-2015].

PUC-Rio - Certificação Digital Nº 1212362/CA

OUSTERHOUT, J. Why threads are a bad idea (for most purposes). In: **USENIX Winter Technical Conference**. [s.n.], 1996. Disponível em: <http://www.cs.utah.edu/ regehr/research/ouster.pdf, http://home.pacbell.net/ouster/threads.ppt>.

SALMITO, T.; RODRIGUEZ, N.; MOURA, A. L. de. **Uma abordagem flexível para o modelo de concorrência em estágios**. Tese (Doutorado) — PUC-Rio, 2013.

SCHMIDT, D. et al. **Pattern-oriented software architecture**. Chichester England New York: Wiley, 2000. ISBN 978-0471606956.

SCHRAGE, L. E.; MILLER, L. W. The queue m/g/1 with the shortest remaining processing time discipline. **Operations Research**, v. 14, n. 4, p. 670–684, 1966. Disponível em: <http://dx.doi.org/10.1287/opre.14.4.670>.

TANESE, R. Distributed genetic algorithms. In: **Proceedings of the 3rd International Conference on Genetic Algorithms**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989. p. 434–439. ISBN 1-55860-066-3. Disponível em: <http://dl.acm.org/citation.cfm?id=645512.657245>.

WELSH, M.; CULLER, D. Adaptive overload control for busy internet servers. In: **Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4**. Berkeley, CA, USA: USENIX Association, 2003. (USITS'03), p. 4–4. Disponível em: <http://dl.acm.org/citation.cfm?id=1251460.1251464>.

WELSH, M.; CULLER, D.; BREWER, E. Seda: An architecture for well-conditioned, scalable internet services. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 35, n. 5, p. 230–243, out. 2001. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/502059.502057>.

ZELDOVICH, N. et al. **Multiprocessor Support for Event-Driven Programs**. 2003.

PUC-Rio - Certificação Digital Nº 1212362/CA

# A
# Modelo de interface

Este capítulo descreve os métodos disponíveis na interface do LEDA. A interface é dividida em 5 grupos, são eles:

1. **Leda**
   Reúne métodos globais da interface, como por exemplo, permitir que as threads visitem a fila privada de cada estágio ao invés de visitarem a fila global.

2. **Pool**
   Reúne métodos de configuração de pools de threads, como por exemplo criar um novo pool ou criar novas threads.

3. **Monitoring**
   Reúne métodos de monitoramento, como por exemplo configurar *callbacks* que são disparadas periodicamente com o objetivo de monitorar a aplicação.

4. **Scheduler**
   Reúne métodos de configuração de filas, como por exemplo configurar a ordem de visitação das threads.

5. **Instances**
   Reúne métodos de configuração de instâncias por estágio, como por exemplo criar e remover novas instâncias.

A divisão deste capítulo é dada pela seguinte forma: demonstraremos todos os métodos disponíveis na interface divididos por cada um dos grupos e por fim, a conclusão do capítulo. A seguir demonstraremos os métodos disponíveis em cada um dos grupos citados acima:

PUC-Rio - Certificação Digital Nº 1212362/CA

## A.1
## Leda

Nesta seção descreveremos os métodos globais disponíveis no grupo *Leda*. Este grupo reúne métodos de configuração das filas da arquitetura, como por exemplo: configurar número máximo de eventos a serem aceitos pela fila de entrada de cada estágio, expor métodos que permitam habilitar e desabilitar uma determinada fila de entrada e permitir a configurar o tipo de fila de processamento, seja privada ou global.

### A.1.1
### int leda.usePrivateQueues()

Este método desabilita a fila global e configura a arquitetura para processar eventos visitando as filas privadas de cada estágio.

**Saída**

| Tipo | Descrição |
|---|---|
| int | 0 em caso de erro e 1 em caso de sucesso |

PUC-Rio - Certificação Digital Nº 1212362/CA

### A.1.2
### int leda.setQueueCapacity (stage_t stage, int maxSize)

Este método configura um limite máximo para o número de eventos acumulados na fila de um estágio.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | Estágio a ter sua capacidade alterada |
| int | maxSize | Número máximo de eventos a serem acumulados |

**Saída**

| Tipo | Descrição |
|---|---|
| int | 0 em caso de erro e 1 em caso de sucesso |

### A.1.3
### int leda.getQueueCapacity (stage_t stage)

Este método captura o tamanho máximo de eventos que podem ser acumulados na fila de entrada de um determinado estágio.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | Estágio que terá sua configuração de número máximo de eventos retornado |

**Saída**

| Tipo | Descrição |
|---|---|
| int | Inteiro indicando o número máximo de eventos que podem ser acumulados na fila de entrada do estágio selecionado |

### A.1.4
### int leda.disableQueue (stage_t stage)

Este método desabilita a fila de entrada de um determinado estágio.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | Estágio que será desabilitado |

**Saída**

| Tipo | Descrição |
|---|---|
| int | 0 em caso de erro e 1 em caso de sucesso |

### A.1.5
### int leda.enableQueue (stage_t stage)

Este método habilita a fila de entrada de um determinado estágio.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | Estágio que será habilitado |

**Saída**

| Tipo | Descrição |
|---|---|
| int | 0 em caso de erro e 1 em caso de sucesso |

PUC-Rio - Certificação Digital Nº 1212362/CA

### A.1.6
### int leda.queueIsEnabled (stage_t stage)

Este método verifica se um estágio está com sua fila de entrada habilitada.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | Estágio que terá sua configuração de fila retornada |

**Saída**

| Tipo | Descrição |
|---|---|
| int | 0 se estiver desabilitado e 1 se estiver habilitado |

## A.2
## Pool

Nesta seção descreveremos os métodos disponíveis no grupo *Pool*. Este grupo reúne métodos de configuração dos pools de threads, como por exemplo: criação de novos pools, captura de pool e criação e remoção de novas threads.

### A.2.1
### pool_t leda.pool.new (int numThreads)

Este método cria um novo pool de threads.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| int | numThreads | número de threads inicial |

**Saída**

| Tipo | Descrição |
|---|---|
| pool_t | pool de threads com "numThreads"threads |

PUC-Rio - Certificação Digital Nº 1212362/CA

### A.2.2
### int leda.pool.setPool (pool_t pool, stage_t stage)

Este método associa um pool a um estágio.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| pool_t | pool | pool que será associado ao estágio informado |
| stage_t | stage | estágio que será configurado com um novo pool |

**Saída**

| Tipo | Descrição |
|---|---|
| int | 0 indicando erro e 1 indicando sucesso |

### A.2.3
### int leda.pool.add (pool_t pool, int numThreads)

Este método cria threads em um pool de threads.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| pool_t | pool | pool de threads que receberá novas threads |
| int | numThreads | número de threads a serem criadas no pool |

**Saída**

| Tipo | Descrição |
|---|---|
| int | 0 indicando erro e 1 indicando sucesso |

### A.2.4
### int leda.pool.kill (pool_t pool, int numThreads)

Este método remove threads de um pool de threads.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| pool_t | pool | pool de threads que terá threads removidas |
| int | numThreads | número de threads a serem removidas do pool |

**Saída**

| Tipo | Descrição |
|---|---|
| int | 0 indicando erro e 1 indicando sucesso |

PUC-Rio - Certificação Digital Nº 1212362/CA

### A.2.5 pool_t leda.pool.getPool (stage_t stage)

Este método captura o pool de threads de um determinado estágio.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | estágio que terá seu pool retornado |

**Saída**

| Tipo | Descrição |
|---|---|
| pool_t | pool de threads do estágio informado |

### A.2.6 int leda.pool.getThreadsCount (pool_t pool)

Este método captura o número de threads em um determinado pool de threads.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| pool_t | pool | estágio que terá seu pool retornado |

**Saída**

| Tipo | Descrição |
|---|---|
| int | número de threads alocadas no pool |

## A.3 Monitoring

Nesta seção descreveremos os métodos disponíveis no grupo *Monitoring*. Este grupo reúne métodos de monitoração da aplicação, como por exemplo: configuração de *callbacks* e captura *throughput*.

PUC-Rio - Certificação Digital Nº 1212362/CA

### A.3.1 void leda.monitoring.addTimer (function i, int seconds)

Este método dispara uma determinada função de tempos em tempos.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| function | i | função que será disparada a cada passo |
| int | seconds | tempo em segundos que a função $i$ será disparada |

**Saída**

| Tipo | Descrição |
|---|---|
| int | número de threads alocadas no pool |

### A.3.2 int leda.monitoring.getCpusCount ()

Este método captura o número total de CPU's da máquina.

**Saída**

| Tipo | Descrição |
|---|---|
| int | número total de CPU's existentes na máquina |

### A.3.3 int leda.monitoring.getCpuUsage ()

Este método captura o uso das CPU's da máquina.

**Saída**

| Tipo | Descrição |
|---|---|
| int | porcentagem de 0 a 100 indicando o uso das CPU's disponíveis na máquina |

PUC-Rio - Certificação Digital Nº 1212362/CA

### A.3.4
### int leda.monitoring.getQueueSize (stage_t stage)

Este método captura a quantidade de eventos acumulados na fila de entrada de um determinado estágio.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | estágio que terá a contagem de sua fila de entrada retornada |

**Saída**

| Tipo | Descrição |
|---|---|
| int | número total de eventos a serem processados |

### A.3.5
### int leda.monitoring.getProcessedCount (stage_t stage)

Este método captura o total de eventos processados de um determinado estágio desde o início de sua execução.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | estágio que terá o total de eventos processados capturado |

**Saída**

| Tipo | Descrição |
|---|---|
| int | Total de eventos processados do estágio selecionado |

### A.3.6
### int leda.monitoring.getInputCount (stage_t stage)

Este método captura o total de eventos que entraram na fila de eventos de um determinado estágio desde o início de sua execução.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | estágio que terá o total de eventos de entrada capturado |

PUC-Rio - Certificação Digital Nº 1212362/CA

**Saída**

| Tipo | Descrição |
|---|---|
| int | Total de eventos de entrada do estágio selecionado |

### A.3.7
### int leda.monitoring.resetStatistics (stage_t stage)

Este método reinicia todas as estatísticas do estágio, como por exemplo o total de eventos processados.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | estágio que terá suas estatísticas reiniciadas |

### A.3.8
### void leda.monitoring.fireLastFocused (function i)

Este método dispara uma determinada função quando o último estágio da ordem de visitação é visitado por uma das threads da aplicação.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| function | i | função que será disparada quando o último estágio for visitado pelas threads |

### A.3.9
### void leda.monitoring.doNotFireLastFocused ()

Este método desabilita a função que é disparada quando as threads visitam o último estágio da ordem de visitação.

## A.4
## Scheduler

Nesta seção descreveremos os métodos disponíveis no grupo *Scheduler*. Este grupo reúne métodos de configuração de processamento de eventos, como por exemplo: configuração da ordem de visitação das threads, priorização de estágios e roubo de threads entre pools de threads.

PUC-Rio - Certificação Digital Nº 1212362/CA

### A.4.1
### void leda.scheduler.visitOrder (stage_t[] stages)

Este método configura a ordem de visitação dos estágios pelas threads.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t[] | stages | lista de estágios na ordem em que serão visitados pelas threads |

### A.4.2
### void leda.scheduler.maxVisits (int maxVisits, function i)

Este método dispara uma função após as threads visitarem um determinado número de estágios.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| int | maxVisits | número de visitas que as threads devem fazer até disparar a função configurada |
| function | i | função que será disparada após um determinado número de visitas realizadas pelas threads |

### A.4.3
### void leda.scheduler.restartAtIndex (int index)

Este método reinicia o processo de visitação das threads por um determinado estágio a cada estágio visitado com pelo menos um evento processado.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| int | index | índice do estágio que será visitado após a visita de um estágio com pelo menos um evento processado de acordo com o *array* de ordem de visitação |

### A.4.4
### void leda.scheduler.doNotRestart ()

Este método desabilita o processo de reinicialização de um estágio quando um estágio é visitado com pelo menos um evento processado.

PUC-Rio - Certificação Digital Nº 1212362/CA

### A.4.5 void leda.scheduler.steal (stage_t stage1, stage_t stage2, int numThreads)

Este método realoca threads entre pools de threads.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage1 | estágio que está requisitando novas threads |
| stage_t | stage2 | número total de threads que deve ser alocado em *stage1* |

### A.4.6 void leda.scheduler.setPriority (stage_t stage, int priority)

Este método prioriza estágios ordenando eventos pela prioridade do estágio na fila global.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | estágio que terá sua prioridade alterada |
| int | priority | prioridade a ser configurada para o estágio de entrada |

### A.4.7 int leda.scheduler.getPriority (stage_t stage)

Este método captura a prioridade de um determinado estágio.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | estágio que terá sua prioridade capturada |

**Saída**

| Tipo | Descrição |
|---|---|
| int | prioridade do estágio |

PUC-Rio - Certificação Digital Nº 1212362/CA

### A.4.8
### void leda.scheduler.maxEventsWhenFocused (stage_t stage, int maxEvents)

Este método configura um número máximo de eventos que as threads podem processar até visitarem o próximo estágio. Disponível apenas através da configuração de processamento de eventos através das filas de entrada de cada estágio.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | estágio que terá sua prioridade capturada - caso "all"seja passado como parâmetro, a configuração se aplicará a todos os estágios |
| int | maxEvents | Número máximo de eventos a serem processados |

PUC-Rio - Certificação Digital Nº 1212362/CA

## A.5
## Instances

Nesta seção descreveremos os métodos disponíveis no grupo *Instances*. Este grupo reúne métodos configuração de instâncias, como por exemplo: criação e remoção de instâncias e captura de instâncias alocadas para um determinado estágio.

### A.5.1
### void leda.instances.add (stage_t stage, int instances)

Este método cria novas instâncias para um determinado estágio.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | estágio que receberá novas instâncias |
| int | instances | número de instâncias a serem criadas para o estágio selecionado |

### A.5.2
### void leda.instances.kill (stage_t stage, int instances)

Este método remove instâncias de um determinado estágio.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | estágio que terá instâncias removidas |
| int | instances | número de instâncias a serem removidas do estágio selecionado |

### A.5.3
### int leda.instances.getInstancesCount (stage_t stage)

Este método captura o número total de instâncias disponíveis em um determinado estágio.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | estágio que terá seu total de instâncias capturado |

**Saída**

| Tipo | Descrição |
|---|---|
| int | total de instâncias existentes para o estágio selecionado |

### A.5.4
### void leda.instances.getFreeInstancesCount (stage_t stage)

Este método captura o total de instâncias livres em um determinado estágio.

**Entrada**

| Tipo | Nome | Descrição |
|---|---|---|
| stage_t | stage | estágio que terá seu total de instâncias livres capturado |

**Saída**

| Tipo | Descrição |
|---|---|
| int | total de instâncias livres do estágio selecionado |

PUC-Rio - Certificação Digital Nº 1212362/CA

## A.6 Discussão

Neste capítulo pudemos ver todos os métodos criados na interface de LEDA que são necessários para a construção de novos controladores. Com esta interface é possível implementar controladores com diferentes comportamentos, possibilitando que desenvolvedores possam construir seus próprios controladores sem que seja necessário entender o código por trás da aplicação. Todos os controladores desenvolvidos neste trabalho foram implementados utilizando a interface de LEDA.

PUC-Rio - Certificação Digital Nº 1212362/CA