



Tiago Lima Salmito

**Uma abordagem flexível para o modelo de
concorrência em estágios**

Tese de Doutorado

Tese apresentada ao Programa de Pós-graduação em Informática
do Departamento de Informática da PUC-Rio como requisito
parcial para obtenção Do título de Doutor em Informática

Orientador : Prof. Noemi de La Rocque Rodriguez
Co-Orientador: Prof. Ana Lúcia de Moura

Rio de Janeiro
Setembro de 2013



Tiago Lima Salmito

**Uma abordagem flexível para o modelo de
concorrência em estágios**

Tese 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 título de Doutor em Informática. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Noemi de La Rocque Rodriguez
Orientador
Departamento de Informática — PUC-Rio

Prof. Ana Lúcia de Moura
Co-Orientador
Departamento de Informática — PUC-Rio

Prof. Markus Endler
Departamento de Informática — PUC-Rio

Prof. Renato Fontoura de Gusmão Cerqueira
Departamento de Informática — PUC-Rio

Prof. Cesar Augusto FonticIELha De Rose
PUC-RS

Prof. Silvana Rossetto
UFRJ

Prof. José Eugênio Leal
Coordenador Setorial do Centro Técnico Científico — PUC-Rio

Rio de Janeiro, 02 de Setembro de 2013

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

Tiago Lima Salmito

Graduou-se em Ciência da Computação, modalidade Computação Científica, na Universidade Federal do Rio Grande do Norte em 2003. Obteve o título de Mestre em Informática pela Universidade Federal da Paraíba em 2007.

Ficha Catalográfica

Salmito, Tiago Lima

Uma abordagem flexível para o modelo de concorrência em estágios / Tiago Lima Salmito; orientadora: Noemi de La Rocque Rodriguez; co-orientadora: Ana Lúcia de Moura. — 2013.

107 f: il. ; 30 cm

1. Tese (doutorado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2013.

Inclui bibliografia

1. Informática – Teses. 2. Modelos de concorrência. 3. Concorrência híbrida. 4. Threads. 5. Orientação a eventos. 6. Estágios. I. Rodriguez, Noemi de La Rocque. II. Moura, Ana Lúcia de. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Para Sílvia Helena.

Agradecimentos

À Noemi por seus conselhos, paciência e amizade;

À Ana Lúcia pela sua dedicação, contribuição e por estar sempre presente;

À PUC-Rio pela oportunidade de desenvolver este trabalho;

À equipe do Laboratório Tecgraf da PUC-Rio por viabilizar os experimentos deste trabalho.

Ao Leonardo Duarte por implementar os algoritmos e pelo fornecimento do conjunto de dados usados na aplicação descrita na seção 5.3.2.

À equipe da Intelie, pelos conselhos e ajuda na validação da plataforma desenvolvida.

Resumo

Salmito, Tiago Lima; Rodriguez, Noemi de La Rocque; Moura, Ana Lúcia de. **Uma abordagem flexível para o modelo de concorrência em estágios**. Rio de Janeiro, 2013. 107p. Tese de Doutorado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O objetivo deste trabalho é explorar e estender a flexibilidade provida pelo modelo híbrido de concorrência orientado a estágios, que visa integrar tanto *loops* de eventos cooperativos como *threads* preemptivas em um conceito único de mais alto nível. As contribuições deste trabalho estão centradas numa proposta de extensão do modelo de estágios que desacopla a especificação de aplicações concorrentes das decisões relacionadas ao ambiente de execução, permitindo que elas sejam flexivelmente mapeadas em diferentes configurações de acordo com as necessidades de escalonamento de tarefas e granularidade de processamento em partes específicas da aplicação. Procurando prover uma definição adequada para o conceito de modelos de concorrência híbridos, propomos um sistema de classificação que se baseia na combinação de características básicas de sistemas concorrentes e a possibilidade de execução paralela em múltiplos processadores. Com base nessa classificação, analisamos os benefícios e desvantagens associados a cada modelo de concorrência, justificando a adoção de modelos que combinam *threads* e eventos em um mesmo ambiente de programação, e descrevemos a extensão do paradigma de programação orientado a estágios. Finalmente, apresentamos a implementação do modelo proposto na linguagem Lua e seu uso em cenários de execução que confirmam os benefícios da extensão do modelo de estágios na especificação de aplicações concorrentes.

Palavras-chave

Modelos de concorrência. Concorrência híbrida. Threads.
Orientação a eventos. Estágios.

Abstract

Salmito, Tiago Lima; Rodriguez, Noemi de La Rocque (Advisor); Moura, Ana Lúcia de. **A flexible approach to staged events**. Rio de Janeiro, 2013. 107p. PhD Thesis — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The purpose of this work is to explore and extend the flexibility provided by the staged event-driven concurrency model to integrate both cooperative event loops and preemptive threads in a single high level abstraction. The contributions of this work are focused on the extension of the staged model that decouples the specification of concurrent applications of the decisions related to the execution environment, allowing them to be flexibly mapped to different configurations according to the task scheduling needs and processing granularity in specific parts of the application. In order to provide an adequate definition of the concept of hybrid concurrency models, we propose a classification system that is based on the combination of basic features of concurrent systems and the possibility of parallel execution on multiple processors. Based on this classification, we analyze the benefits and drawbacks associated with each concurrency model, justifying the adoption of models that combine threads and events in the same programming environment and the extension of the staged model. Finally, we present the implementation of the proposed model in the Lua programming language and its use in execution scenarios that confirm the benefits of the extension of the staged model in the specification of concurrent applications.

Keywords

Concurrency models. Hybrid concurrency. Threads. Event-driven. Stages.

Sumário

1	Introdução	13
1.1	Objetivos e Organização do Trabalho	16
2	Modelos de Concorrência Híbridos	18
2.1	Classificação de Modelos de Concorrência Híbridos	19
2.1.1	Concorrência Híbrida Orientada a Eventos	21
2.1.2	Concorrência Híbrida Baseada em Threads	23
2.1.3	Concorrência Híbrida em Estágios	26
2.2	Discussão	30
3	Uma Extensão do Modelo de Concorrência em Estágios	35
3.1	Modelo de Programação	35
3.2	Estágios	38
3.3	Comunicação	39
3.4	Aglomeramento de Estágios	41
3.5	Mapeamento de Aglomerados	43
3.6	Discussão	45
4	Leda: Uma Implementação do Modelo de Estágios em Lua	47
4.1	Ambiente de Programação	48
4.1.1	Abstrações de Programação Leda	48
4.1.2	A API Leda para Programação de Estágios	50
4.1.3	Construção do Grafo de Estágios	54
4.1.4	Aglomeramento, Mapeamento e Execução	55
4.2	Comunicação e Gerenciamento de Dados	57
4.3	Implementação de Leda	58
4.3.1	Gerenciamento de Instâncias	59
4.3.2	Processo Leda	61
5	Avaliação	65
5.1	Flexibilidade do Modelo	65
5.1.1	Pipeline de Processamento de Imagens	66
5.1.2	Processamento Complexo de Fluxos de Eventos	70
5.2	Sobrecarga da Implementação	72
5.2.1	Servidor HTTP de Páginas Estáticas e Dinâmicas	73
5.2.2	Ordenação de Vetores em Memória Distribuída	80
5.3	Facilidade do Uso de Leda	82
5.3.1	Visualização de Ações da Bolsa de Valores	83
5.3.2	Simulação de Tubos Ascendentes de Petróleo	85
6	Conclusão	89
6.1	Sumário de Contribuições	90
6.2	Trabalhos Futuros	91
	Referências Bibliográficas	92

A	API Leda	98
A.1	A API Principal de Leda	98
A.2	A API de Instâncias de Estágios	101
B	Códigos Fonte	103
B.1	Teste de páginas dinâmicas do servidor HTTP	103
B.2	Ordenação Distribuída de Vetores	103

Lista de figuras

2.1	Extensão do gráfico proposto por Adya <i>et al.</i> de forma a contemplar ambientes de execução multiprocessados.	20
2.2	Representação do modelo de concorrência híbrido orientado a eventos.	21
2.3	Representação do modelo de concorrência híbrido baseado em <i>threads</i> .	24
2.4	Representação do modelo de concorrência híbrido em estágios.	27
2.5	Componentes internos de um estágio da arquitetura SEDA.	27
2.6	Estrutura simplificada de um servidor <i>web</i> de páginas estáticas orientado a estágios.	28
2.7	A mesma especificação de uma aplicação em estágios com diferentes granularidades.	32
3.1	Etapas do processo de desenvolvimento PCAM.	36
3.2	Representação gráfica de uma aplicação contendo quatro estágios, suas portas e seus conectores.	37
3.3	Uma possível configuração de aglomeração do grafo da Figura 3.2.	37
3.4	Representação gráfica de um estágio em execução com duas portas de saída.	39
3.5	Grafos de estágios estruturados em três padrões de comunicação diferentes.	40
3.6	Exemplo de grafos equivalentes para uma aplicação.	41
3.7	Mapeamento de aglomerados de estágios em unidades de execução independentes.	44
4.1	Etapas de desenvolvimento Leda.	49
4.2	Representação gráfica da aplicação definida na Listagem 4.4.	55
4.3	Diagrama de transição de estados de instâncias.	60
4.4	Componentes internos de um processo Leda.	62
5.1	Grafo de estágios da aplicação de processamento de imagens e a divisão de aglomerados em três cenários de execução.	67
5.2	Tempos totais de execução da aplicação de processamento de imagens nos casos de teste com imagens de baixa, média e alta resolução de acordo com os cenários de execução com 1, 2 e 3 processos.	68
5.3	Estatísticas de execução da aplicação de processamento de imagens para o caso de teste com imagens de média resolução.	69
5.4	Particionamento do cenário 1 para o grafo de estágios da aplicação de processamento complexo de eventos.	71
5.5	Particionamento do cenário 2 para o grafo de estágios da aplicação de processamento complexo de eventos.	71
5.6	Estrutura do grafo de estágios do servidor HTTP em estágios.	73
5.7	Comparação da vazão de requisições atendidas dos servidores <i>web</i> com páginas estáticas e arquivos de 300kb.	75

5.8	Comparação da latência média de resposta dos servidores <i>web</i> com páginas estáticas e arquivos de 300kb.	76
5.9	Comparação da vazão de requisições atendidas dos servidores <i>web</i> com páginas estáticas e arquivos de 5mb.	77
5.10	Comparação da latência média de resposta dos servidores <i>web</i> com páginas estáticas e arquivos de 5mb.	78
5.11	Comparação da vazão de requisições atendidas dos servidores <i>web</i> com páginas dinâmicas.	78
5.12	Comparação da latência média de resposta dos servidores <i>web</i> com páginas dinâmicas.	79
5.13	Estrutura do grafo de estágios da aplicação distribuída de ordenação de vetores.	80
5.14	Captura de tela da aplicação de simulação de tubos ascendentes de petróleo.	88

Lista de tabelas

5.1	Tamanho das imagens e a tamanho total das 500 imagens usadas no teste.	67
5.2	Tempos totais de execução da aplicação de processamento complexo de eventos.	72
5.3	Comparação dos tempos de execução de Leda e MPI para aplicação distribuída de ordenação de vetores.	81
5.4	Tempos de comunicação de Leda e MPI entre processos remotos.	82

1

Introdução

Duas estratégias têm sido tipicamente usadas para modelar sistemas inerentemente concorrentes: concorrência baseada em *threads* preemptivas e concorrência orientada a eventos. Na concorrência baseada em *threads* preemptivas, as tarefas concorrentes, ou *threads*, podem ser interrompidas antes do término pelo escalonador do sistema operacional, possibilitando a execução simultânea de várias tarefas. Em sistemas orientados a eventos, um único *loop* de eventos invoca sequencialmente tratadores de eventos para processar eventos acumulados em uma fila global, portanto, há apenas um fluxo de execução presente, que é compartilhado cooperativamente pelos tratadores.

Ambas as abordagens têm seus defensores e detratores. O modelo de concorrência baseado em *threads* preemptivas com compartilhamento de memória se tornou o padrão “de fato” para programação de tarefas concorrentes em linguagens como Java (Lea2000), C# (Benton2004) e Python (vanRossum1995). O mecanismo de preempção, disponível na maioria dos sistemas operacionais atuais, livra o desenvolvedor de escalonar manualmente as tarefas, dando uma visão de vários fluxos de controle sequenciais sem interrupção. Entretanto, a programação com *threads* não é uma tarefa fácil. O uso de recursos compartilhados deve ser coordenado, pois o acesso simultâneo a dados compartilhados pode levar a inconsistências no estado interno da aplicação. Travas, semáforos e monitores são mecanismos bem conhecidos para promover a sincronização, criando um acesso exclusivo a regiões críticas do sistema (Silberschatz2009).

O modelo orientado a eventos foi apontado como uma boa alternativa para especificar tarefas concorrentes (Ousterhout1996, Lee2006). Neste modelo, uma tarefa é dividida em um ou mais tratadores de eventos e um *loop* de eventos fica responsável por executar os tratadores à medida em que os eventos são recebidos, fazendo com que o controle só retorne ao *loop* quando o processamento do tratador termina. Assim, tratadores de eventos tipicamente compartilham um único fluxo de execução de forma cooperativa para processar eventos distintos. Alguns problemas associados à concorrência baseada em *threads* são evitados com a utilização de um único *loop* de eventos. Porém, seu uso traz limitações inexistentes no modelo de *threads* preemptivas. Como há

apenas um *loop* de eventos em execução, em arquiteturas com multi-núcleos, apenas um núcleo é usado. Além disso, tratadores muito complexos podem monopolizar o processamento, impedindo que outros eventos sejam tratados. Chamadas bloqueantes também devem ser evitadas em favor do uso de interfaces assíncronas.

O debate sobre as vantagens e desvantagens de cada modelo é recorrente e, normalmente, o desenvolvedor precisa escolher um dos modelos e abdicar das vantagens oferecidas pelo outro. Durante a última década, dois fatores impulsionaram o interesse em modelos de concorrência que permitem a coexistência de *threads* preemptivas e eventos em um mesmo ambiente de execução. O primeiro deles foi a forte adoção de arquiteturas multiprocessadas e a ubiquidade de processadores com arquitetura multi-núcleos, que estimularam a criação de mecanismos para incorporar paralelismo ao gerenciamento cooperativo de tarefas usado pelo modelo orientado a eventos (Dabek2002, Gaud2010). O segundo foi a popularização de cenários de aplicação que demandam concorrência massiva, incentivando a definição de métodos alternativos para reduzir a sobrecarga de controle de execução de um grande número de tarefas simultâneas (vonBehren2003b, Haller2007).

Adya *et al.* (Adya2002) colocaram as diferenças entre os modelos de concorrência (*threads* e eventos) em termos da técnica de gerenciamento de tarefas concorrentes aplicada pelo escalonador (preemptivo ou cooperativo) e do gerenciamento do estado local entre tarefas concorrentes (automático ou manualmente tratado pelo programador). Eles concluem que esses conceitos são ortogonais e, na verdade, podem ser combinados de várias formas para oferecer um ambiente de execução híbrido no desenvolvimento de aplicações concorrentes.

Idealmente, em um modelo de concorrência híbrido, o programador projeta partes de uma aplicação usando *threads*, onde esta abstração é mais apropriada, por exemplo, para representação de fluxos de controle de diferentes clientes em servidores, e partes da aplicação usando eventos, por exemplo, para permitir o uso de interfaces assíncronas de entrada e saída. Diversos sistemas implementam o modelo de concorrência híbrida de várias formas (Welsh2001, Adya2002, Li2007, Haller2007), porém muitas dessas soluções apresentam um viés para um dos modelos de concorrência, que é oferecido ao programador, e o outro é utilizado apenas internamente em sua implementação.

Um obstáculo à incorporação de modelos híbridos de concorrência em um sistema é a dificuldade de implementá-los de forma eficiente. Essa dificuldade é, em grande parte, resultante da necessidade de prover mecanismos independentes para cada modelo de concorrência e meios para que aplicações possam cha-

vear entre os estilos de programação sem introduzir grande sobrecarga adicional de processamento e complexidade de especificação e manutenção de aplicações. Essas exigências necessitam que abstrações de programação sejam introduzidas para lidar com a integração de comunicações assíncronas e múltiplas *threads* em um mesmo ambiente de execução.

A arquitetura de eventos em estágios, ou SEDA (do inglês *Staged Event-Driven Architecture*), é um modelo híbrido voltado para a implementação de sistemas concorrentes que não apresenta um viés claro para nenhum dos modelos de concorrência (Welsh2001). Nesse modelo, *threads* preemptivas são usadas como mecanismo para execução de tarefas concorrentes, enquanto um modelo de comunicação orientado a eventos é oferecido em um nível de abstração superior.

Uma aplicação baseada nessa arquitetura é dividida em módulos chamados de *estágios*, que implementam partes da sua lógica de processamento, cuja execução prossegue de estágio em estágio como um *pipeline*. Estágios possuem internamente um *pool* de *threads* exclusivo e se comunicam através de eventos. Cada estágio possui uma fila de eventos para acumular os eventos a serem processados. Durante o processamento, um estágio pode enfileirar novos eventos na fila de outros estágios, caso seja necessário passar um resultado adiante. Assim, uma aplicação em estágios pode ser visualizada como um grafo direcionado, onde as arestas representam fluxos de eventos entre estágios.

Apesar dos benefícios do uso do modelo de estágios no projeto de aplicações concorrentes, ainda existem algumas questões em aberto que devem ser abordadas durante a divisão de uma aplicação em um conjunto de estágios concorrentes. Em especial, citamos duas questões relevantes: a divisão dos recursos disponíveis no ambiente de execução entre os estágios da aplicação e a granularidade de processamento dos estágios.

Os estágios são implementados como componentes independentes, associados naturalmente aos seus próprios parâmetros de escalonamento. O desempenho geral da aplicação está relacionado ao equilíbrio na divisão de recursos oferecidos pelo ambiente de execução entre os estágios através da configuração de seus parâmetros de escalonamento.

Embora SEDA proporcione o uso de heurísticas para ajuste fino da configuração desses parâmetros e evitar o uso excessivo de recursos durante situações de alta demanda, a granularidade das decisões de escalonamento é fortemente acoplada à estrutura dos estágios da aplicação. O programador pode optar por definir um número maior de estágios de menor complexidade para obter uma granularidade mais fina de gerenciamento de tarefas ou agrupar estágios subsequentes e obter um controle de concorrência com maior

granularidade. Os benefícios da divisão da aplicação em estágios cada vez menos complexos são diminuídos devido ao excesso de trocas de contexto entre as *threads* de estágios e o aumento do número de enfileiramentos sucessivos de eventos, uma vez que os estágios são tratados como unidades de escalonamento independentes.

Alguns trabalhos mais recentes procuram tratar esse problema através de métodos que ajustam os parâmetros de execução de acordo com o estado conjunto de todos os estágios da aplicação (Li2006), ou permitindo que estágios compartilhem *threads* cooperativamente (Gordon2010). Porém, esses trabalhos normalmente não adotam soluções específicas para desacoplar as decisões de escalonamento que dependem das características do ambiente de execução e as decisões relacionadas à semântica de funcionamento da aplicação. Nós acreditamos que esse desacoplamento é de fundamental importância para alcançar um bom equilíbrio de divisão de recursos entre partes concorrentes da aplicação, e promover maior flexibilidade no desenvolvimento de aplicações orientadas a estágios.

1.1

Objetivos e Organização do Trabalho

Este trabalho propõe uma extensão do modelo de estágios em que a estrutura de uma aplicação em estágios é definida através da descrição declarativa de um grafo que liga estágios projetados de forma independente por meio de estruturas chamada de *conectores*. Depois, para configurar o ambiente de execução da aplicação, os estágios que compõem o grafo são agrupados em *aglomerados* que são mapeados a processos do sistema operacional, e que podem ser executados em diferentes máquinas. Decisões de escalonamento distintas e configuráveis, como número de *threads* alocadas e mecanismos de prioridade, podem ser aplicadas a cada aglomerado individual. Assim, o programador pode explorar o desacoplamento entre a lógica da aplicação e sua configuração de execução para experimentar com diferentes configurações de composição de estágios e o mapeamento para ambientes de execução distintos sem a necessidade de alterar a estrutura geral da aplicação.

Propomos, no Capítulo 2, um novo sistema de classificação para modelos híbridos de concorrência baseado na abstração de concorrência que é oferecida ao programador. Esse sistema de classificação nos permite distinguir as diversas implementações de concorrência híbrida a partir da combinação de três características básicas: a política de gerenciamento de tarefas (preemptiva ou cooperativa), a política de gerenciamento de pilha local (manual ou automático) e a possibilidade de execução paralela em múltiplos processadores.

Com base nessa classificação, analisamos os benefícios e desvantagens associados a cada modelo de concorrência, justificando a adoção de modelos híbridos de concorrência que combinam *threads* e programação orientada a eventos e a extensão do paradigma de programação orientado a estágios.

Baseados em aspectos levantados durante a classificação de sistemas híbridos, descrevemos, no Capítulo 3, um modelo de concorrência que estende os conceitos originais da arquitetura em estágios para desacoplar a especificação de aplicações de decisões relacionadas ao ambiente de execução, permitindo que elas sejam flexivelmente mapeadas em diferentes configurações de acordo com necessidades de escalonamento de tarefas aplicadas a seu ambiente de execução e granularidade de processamento e comunicação em partes específicas da aplicação.

O Capítulo 4 descreve a implementação do modelo proposto na linguagem de programação Lua e, no Capítulo 5, avaliamos o impacto do uso da implementação em alguns cenários de aplicações concorrentes que exploram a flexibilidade do modelo e exercitam os mecanismos de comunicação e compartilhamento de dados do modelo proposto. Finalmente, no Capítulo 6, apresentamos as conclusões deste trabalho.

2

Modelos de Concorrência Híbridos

Ousterhout (Ousterhout1996) e von Behren *et al.* (vonBehren2003a) são os autores mais citados nos dois lados do debate entre as vantagens e desvantagens de modelos de concorrência baseados em *threads* ou orientados a eventos.

Ousterhout alega que sistemas orientados a eventos são mais adequados na maior parte dos casos, pois como não há concorrência entre os tratadores de evento, não existe a necessidade de se preocupar com sincronização ou *deadlocks*. Além disso, ele afirma que os eventos normalmente são mais rápidos do que *threads* em processadores individuais, considerando que não há trocas de contexto ou sobrecarga de sincronização. Por outro lado, Behren *et al.* defendem a utilização de múltiplas *threads*. Eles argumentam que muitas dessas vantagens devem-se ao modelo de multitarefa cooperativa em que os eventos são baseados, que também pode ser usado como base para a implementação de *threads* em nível de usuário. Outro ponto levantado é que a programação baseada em eventos tende a ofuscar o fluxo de controle de aplicações, efeito que eles chamam de *inversão de controle*.

Apesar das críticas, Ousterhout afirma explicitamente que *threads* não devem ser abandonadas, e realça que são necessárias para se alcançar paralelismo real em arquiteturas multiprocessadas. De fato, a crescente adoção de arquiteturas multi-núcleo e a popularização de cenários de aplicação que demandam concorrência massiva têm levado diversos autores a propor modelos de concorrência híbridos, ou modelos de concorrência que permitem a coexistência de *threads* e eventos no ambiente de execução de uma aplicação.

Este capítulo discute alguns trabalhos que envolvem propostas de modelos de concorrência híbridos. A próxima seção propõe uma classificação para essa classe de modelos de concorrência e a categorização de abordagens recentes que se enquadram neste tema de acordo com a classificação proposta. Em seguida, a partir da análise das características e das deficiências de cada abordagem, discutiremos algumas questões que permanecem em aberto e são desejáveis no projeto de modelos de concorrência híbridos. Essas questões servirão como base para a definição posterior de um modelo de concorrência que

leva em consideração os aspectos identificados.

2.1

Classificação de Modelos de Concorrência Híbridos

Nesta seção, estamos interessados em classificar ambientes de execução que apresentam concorrência híbrida capazes de executar em arquiteturas multiprocessadas (ou SMP, do inglês *Symmetric MultiProcessing*) e arquiteturas multi-núcleo. Para isso, nos valem dos critérios adotados em (Adya2002) para definir uma classificação de modelos de concorrência que exploram o paralelismo real presente nestas arquiteturas.

Adya *et al.* discutem as vantagens e desvantagens da programação baseada em *threads* e a orientação a eventos, considerando a ortogonalidade de conceitos como gerenciamento de pilha manual ou automático, e gerenciamento de tarefas preemptivo ou cooperativo. Os autores argumentam que é benéfico usar o gerenciamento de tarefas cooperativo, onde as tarefas devem passar cooperativamente o controle para o escalonador de tarefas, enquanto se preserva o gerenciamento da pilha automático, em que o estado local da tarefa é preservado durante toda a sua execução. Porém, a solução híbrida dada por eles somente abrange ambientes de execução com um único processador.

A Figura 2.1 propõe uma extensão do gráfico apresentado por Adya *et al.* de forma a contemplar ambientes de execução multiprocessados. O plano (a) apresenta o gráfico em ambientes com processador único, tal como foi proposto em (Adya2002): a região “*multithread*” descreve multitarefa preemptiva com gerenciamento de pilha automático, como é o caso de *threads* de sistema operacional. Na região “orientação a eventos”, os tratadores de eventos em execução devem ceder o controle voluntariamente para o escalonador, exigindo esforço manual pelo programador para gerenciar a perda de estado local entre diferentes invocações de tratadores de eventos. A região “*threads* cooperativas” representa modelos de concorrência que usam o gerenciamento de tarefas cooperativo, no qual a informação da pilha local é automaticamente encapsulada e mantida através de uma abstração de *thread* em nível de usuário, por exemplo, através de co-rotinas (deMoura2009) e *closures* ou continuacões (Li2007, Sussman1998).

O plano (b) da Figura 2.1 apresenta as três representações originais projetadas em seus pares em um ambiente multiprocessado. Nessa classificação, a região “*multithread*” mantém as mesmas características de antes, exceto pelo fato de que agora *threads* podem ser executadas com paralelismo real. Estamos interessados nas duas regiões à esquerda: ambas representam sistemas híbridos que combinam características de *threads* preemptivas e eventos, porém cada

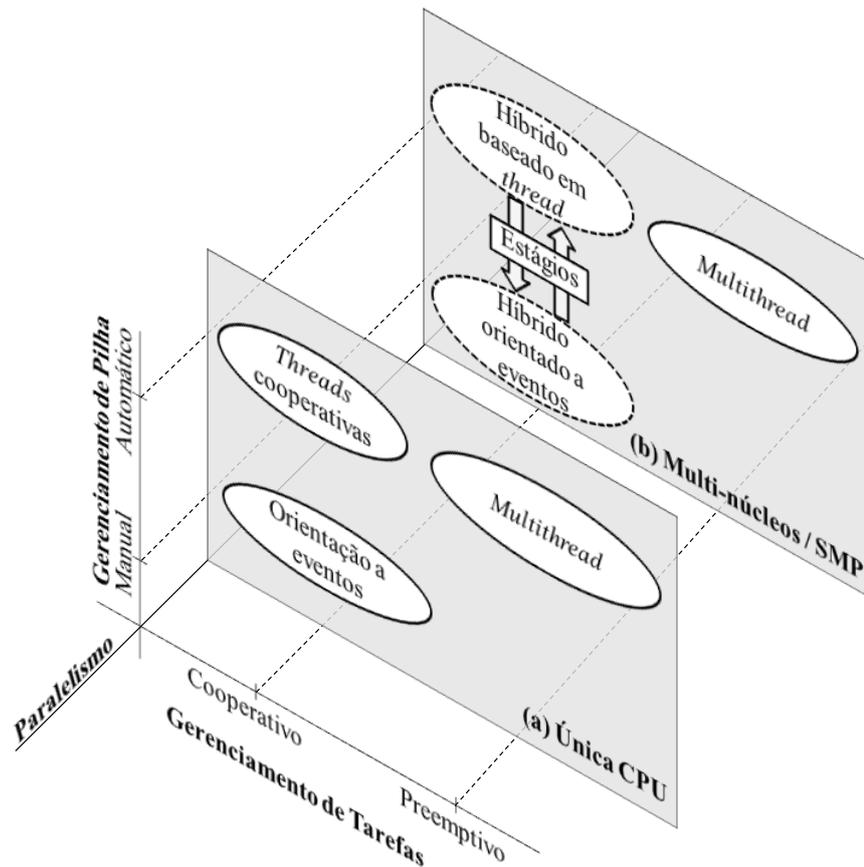


Figura 2.1: Extensão do gráfico proposto por Adya *et al.* de forma a contemplar ambientes de execução multiprocessados.

qual apresenta um viés diferente para uma das abstrações. Nós vamos usar esse viés como base para identificar, compreender e analisar as diferentes classes de modelos de concorrência híbridos.

O modelo de concorrência híbrido orientado a eventos é o resultado da extensão do modelo orientado a eventos clássico, com única *thread*, para ambientes multiprocessados através da presença de múltiplos *loops* de eventos executando em paralelo. Por exemplo, alguns sistemas executam *loops* de eventos independentes em *threads* separadas para evitar que programas baseados em eventos fiquem bloqueados em chamadas síncronas, ou as criam sob demanda ao serem confrontados com operações bloqueantes (Pai1999). Opostamente, o modelo híbrido baseado em *threads* mantém a combinação de multitarefa cooperativa e gerenciamento de pilha automático, provido por *threads* em nível de usuário, porém executando em ambientes com múltiplos processadores através do mapeamento $N \times M$ com *threads* de sistema operacional. Nesses dois casos, apenas um modelo de concorrência é apresentado para o programador e o outro é utilizado como auxílio em sua implementação.

A terceira classe de modelos de concorrência híbridos identificada, o modelo orientado a estágios, não apresenta um viés claro para *threads* ou

eventos, e expõe as duas abstrações para o programador. Sistemas baseados neste modelo são tipicamente decompostos em módulos independentes que se comunicam apenas através de filas de eventos. Internamente, os módulos usam múltiplas *threads* para permitir o consumo de eventos de forma concorrente. Esta classe de sistemas é inspirada na arquitetura SEDA (Welsh2002), na qual os módulos são chamados de estágios. No plano **(b)** da Figura 2.1, esta alternativa é representada pelo retângulo que se encontra entre os modelos híbridos baseados em *threads* e os modelos híbridos orientados a eventos.

Nas próximas sub-seções, discutiremos cada um dos modelos de concorrência híbridos identificados em mais detalhes, fornecendo alguns exemplos de sistemas que se enquadram em cada categoria.

2.1.1

Concorrência Híbrida Orientada a Eventos

O modelo de concorrência híbrido orientado a eventos enfatiza as vantagens da programação orientada a eventos, principalmente relacionadas com a gerenciamento de tarefas cooperativo em nível de usuário, porém eliminando-se a limitação de trabalhar apenas com um único processador. Para suportar paralelismo real, soluções situadas nessa classe utilizam dois ou mais *loops* de eventos concorrentes. A Figura 2.2 contém uma representação conceitual desse modelo.

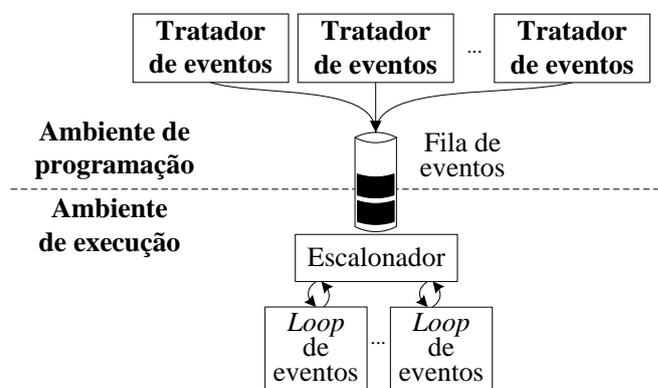


Figura 2.2: Representação do modelo de concorrência híbrido orientado a eventos.

A presença de paralelismo entre os tratadores de eventos traz o problema de sincronização de acesso ao estado compartilhado novamente à tona. Se vários tratadores de eventos executarem simultaneamente com acesso a uma memória global compartilhada, o programador terá que lidar com os problemas clássicos de condições de corrida. Modelos híbridos orientados a eventos normalmente tentam evitar esse cenário controlando o acesso à memória global de diversas maneiras.

Conceitualmente, os tratadores de eventos paralelos podem ser executados em processos de sistema operacional exclusivos, eliminando-se assim o problema de memória compartilhada, no entanto, nós estamos interessados em propostas de sistemas que apresentam concorrência híbrida no qual a execução de tratadores de eventos se dá dentro de um mesmo espaço de endereçamento, o que elimina a possibilidade de execução em ambientes com memória distribuída.

Com relação ao escalonamento de tarefas em nível de usuário, grande parte das abordagens híbridas orientadas a eventos recorrem a um *pool* de *threads*, onde cada *thread* executa um *loop* de eventos concorrente para invocar os tratadores de eventos definidos no ambiente de programação. Isto é interessante na medida em que permite que o ambiente de execução controle o paralelismo real envolvido, por meio da configuração do número de *threads* de sistema operacional que será usado. A presença da camada de escalonamento também implica em uma maior portabilidade do que o uso direto de *threads* de sistema operacional no ambiente de programação.

A seguir, descreveremos duas propostas de sistemas que apresentam o modelo de concorrência híbrido orientado a eventos: a biblioteca Libasync-smp (Dabek2002) e o modelo de eventos InContext (Yoo2011).

A biblioteca Libasync-smp (Dabek2002), uma extensão da biblioteca Libasync (Mazieres2001) para aplicações em C e C++, é um exemplo de abordagem para concorrência híbrida orientada a eventos que provê paralelismo através da execução simultânea de diferentes tratadores de eventos em múltiplos processadores.

Um programa que faz uso dessa biblioteca é tipicamente um processo de sistema operacional com uma *thread* por processador (ou núcleo) disponível. Cada uma dessas *threads* repetidamente executa *callbacks* (tratadores de eventos), obtidas de uma fila de *callbacks* prontas para executar. Todas as *threads* compartilham o estado global do processo, isto é, variáveis globais, descritores de arquivos abertos, etc. Para coordenar o acesso ao ambiente compartilhado, a biblioteca introduz a noção de coloração de eventos. Cada *callback* definida pela aplicação está associada a uma cor (um valor de 32 bits), de forma análoga a uma trava de exclusão mútua: *callbacks* associadas à mesma cor não podem ser executadas em paralelo. Por padrão, todas as *callbacks* são mutuamente exclusivas ou seja, elas têm a mesma cor. Isso estimula o programador a introduzir paralelismo de forma incremental dentro do ambiente de execução da aplicação.

InContext (Yoo2011) é uma extensão para o sistema orientado a eventos MACE (Killian2007), no qual tratadores de eventos são atômicos e executados

em uma única *thread*. Em uma aplicação InContext, o paralelismo é alcançado através da atribuição de tratadores de eventos pendentes a *threads* de sistema operacional que executam *loops* de eventos concorrentes. Tratadores de eventos têm acesso a memória compartilhada e, para coordenar o acesso ao estado global da aplicação, a biblioteca introduz a noção de contextos com diferentes níveis de permissão de acesso. InContext define três tipos de contextos: *none*, *anon* e *global*.

Um tratador de eventos pode transitar entre os diferentes níveis de contexto durante sua execução, porém suas ações são limitadas por seu contexto atual de execução. Os tratadores de eventos que executam no contexto *none* não podem acessar quaisquer dados globais. Tratadores no contexto *anon* apenas podem efetuar operações de leitura de dados globais. O contexto *global* permite leituras e alterações no estado global da aplicação.

O ambiente de execução InContext garante que apenas um tratador de eventos está ativo no contexto *global* e que um tratador de eventos não entre no contexto *global* enquanto algum tratador esteja em execução no contexto *anon* ou *global* através de uma trava de leituras e escritas. O programador deve anotar as transições entre contextos manualmente junto ao código do tratador de eventos, e cabe ao programador garantir que os acessos a variáveis globais sejam efetuados apenas no contexto apropriado.

2.1.2 Concorrência Híbrida Baseada em Threads

Modelos de concorrência híbridos baseados em *threads* correspondem à extensão do modelo de multitarefa cooperativa com gerenciamento de pilha automático aplicada a ambientes multiprocessados. As propostas enquadradas nessa classe expõem abstrações de *threads* ao programador que são executadas através de uma implementação em nível de usuário com base em eventos (vonBehren2003b). A Figura 2.3 mostra uma representação gráfica dos componentes desse modelo.

Nesta classe de modelos de concorrência híbrida, *threads* de usuário cooperativas executam sobre múltiplas *threads* de sistema operacional, e, como é o caso de *threads* cooperativas que executam em ambientes com processador único, chamadas de sistema que bloqueiam devem ser traduzidas em chamadas assíncronas (eventos) através do uso de *wrappers* (ou envelopes). O fluxo de controle é cooperativamente passado ao escalonador, que também executa em nível de usuário, e o estado da *thread* de usuário é mantido enquanto sua execução está suspensa, através de estruturas de dados que encapsulam o estado atual. Após a recepção do evento que indica que a operação solicitada

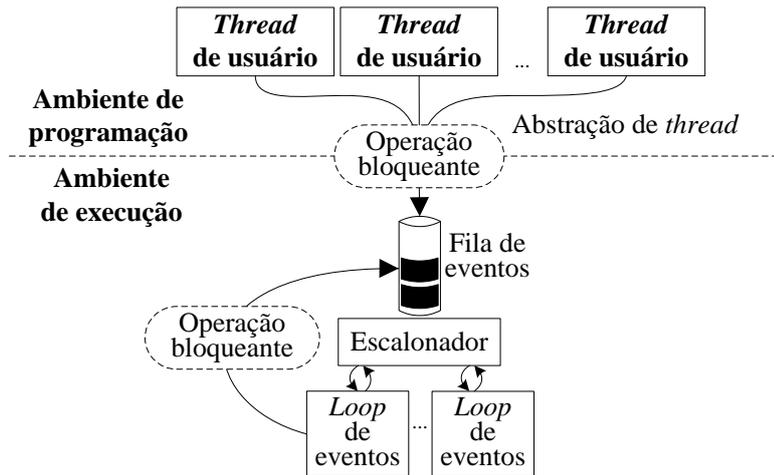


Figura 2.3: Representação do modelo de concorrência híbrido baseado em *threads*.

foi concluída, o escalonador restaura o estado da *thread* de usuário e retoma sua execução, escondendo do programador a inversão de controle típica do modelo orientado a eventos. Para fazer com que a inversão do controle seja transparente, estes modelos podem recorrer a continuções (Li2007), ou a co-rotinas e *closures* (vonBehren2003b, Silvestre2010).

A existência de múltiplos processadores pode ser explorada através da execução de um número arbitrário de *loops* de eventos concorrentes, que constantemente consomem eventos da fila. Em alguns casos, *loops* de eventos exclusivos lidam com operações específicas (como chamadas de sistema bloqueantes ou lidar com operações de rede) e possuem filas de eventos próprias.

Implementações de *threads* de usuário permitem o uso de políticas de escalonamento flexíveis e evitam a sobrecarga de troca de contexto do sistema operacional. Porém, quando são executadas em múltiplas *threads* de sistema operacional, elas estão sujeitas a condições de corrida caso executem em ambientes com estado compartilhado.

Apresentamos a seguir, três implementações que empregam o modelo de concorrência híbrido baseado em *threads*: a biblioteca Capriccio (vonBehren2003b), a biblioteca de atores para Scala (Haller2007), e a proposta de Li e Zdancewic (Li2007).

Capriccio (vonBehren2003b) é uma biblioteca para programas em C que fornece uma API compatível com POSIX (IEEE/ANSI1996) para *threads* cooperativas em nível de usuário. Nesta solução, a multitarefa cooperativa é implementada através de co-rotinas (Toernig2000). Capriccio fornece *wrappers* para operações bloqueantes da API padrão C para chamar versões assíncronas

homólogas e ceder transparentemente o controle para o escalonador.

Uma característica do escalonador proposto por Capriccio é a maneira com que ele explora a flexibilidade do escalonamento em nível de usuário. O sistema coleta informações sobre o uso de recursos e de frequência de bloqueio de cada *thread* de usuário ao longo de sua execução. O escalonador pode usar esta informação para implementar políticas diferenciadas, como dar prioridade às *threads* de usuário que estão mais próximas de terminar ou que estão mais propensas a liberar recursos.

O modelo de concorrência orientado a atores (Agha1985) descreve uma aplicação distribuída como um conjunto de objetos independentes que interagem através de mensagens assíncronas. A biblioteca de atores para Scala (Haller2007, Odersky2008) provê suporte a esse paradigma de duas formas alternativas. A primeira é através da primitiva `receive`, que bloqueia a *thread* atual até que alguma mensagem esteja disponível para consumo. A segunda forma é através da primitiva não bloqueante `react`, que encapsula o código a ser executado no momento da chegada da mensagem através de uma continuação da computação em execução. Essa continuação representa o ator suspenso, e é retomada apenas quando uma mensagem se torna disponível.

Novamente, *loops* de eventos concorrentes permitem explorar os recursos de ambientes multiprocessados. Tarefas são geradas pela criação de novos atores ou pelo pareamento de primitivas `send` e `receive` e pela primitiva `react`. Neste último caso, a nova tarefa é um *closure* que contém a continuação do ator que irá tratar a mensagem. A execução de uma tarefa é concluída quando seu ator associado termina ou quando se executa a primitiva `react`.

Como os atores em Scala são implementados como objetos Java, o cuidado para que eles se comuniquem apenas através de troca de mensagens é deixado para o programador (Odersky2008). Referências a objetos passadas em mensagens podem ser compartilhadas entre os atores que, portanto, estarão sujeitos a problemas de compartilhamento de memória em ambientes com múltiplas *threads*. Esses problemas devem ser tratados através de técnicas clássicas para controle de exclusão mútua.

O trabalho desenvolvido por Li e Zdancewic (Li2007) explora os recursos da linguagem de programação Haskell para implementar uma biblioteca de suporte a multitarefa cooperativa capaz de executar em múltiplos processadores. Nessa abordagem, o código da aplicação é escrito sequencialmente em uma abstração de *threads* em nível de usuário através da sintaxe ‘do’ fornecida por Haskell (Haskell2013).

Cada *thread* de usuário é convertida internamente para uma sequência de etapas delimitadas por chamadas de sistema, ou pontos considerados como

potencialmente bloqueantes. Uma fila de tarefas prontas contém continuações que representam eventos a serem tratados pelos *loops* de eventos concorrentes, que repetidamente retiram uma dessas continuações prontas, executando-as até a próxima chamada do sistema. Dependendo de sua natureza, a chamada de sistema pode ser expedida a um *loop* de eventos exclusivo. Quando os resultados da chamada de sistema estão disponíveis, a continuação atualizada da tarefa é colocada na fila de tarefas prontas. A execução dos passos sequenciais que formam as *threads* de usuário são controladas através da característica de avaliação atrasada (ou *lazy*) de Haskell.

Essa abordagem é bastante similar à usada pelo mecanismo de *threads* de usuário de Capriccio, com a particularidade de que continuações, em vez de rotinas, escondem a inversão de controle. Li e Zdancewic utilizam o conceito de mônadas (Moggi1991) oferecido por Haskell para representar as continuações e encapsular o estado local da computação de uma *thread* de usuário.

O sistema pressupõe que todos os passos sequenciais das *threads* de usuário são isolados e podem ser realizadas em paralelo. Isto é facilitado pelo fato de Haskell ser uma linguagem funcional pura. Entradas, saídas ou quaisquer outras operações em recursos compartilhados podem ser tratadas através de *loops* de eventos sequenciais exclusivos, ou por primitivas de sincronização explícitas como travas de exclusão mútua em nível de usuário.

2.1.3 Concorrência Híbrida em Estágios

O modelo de concorrência híbrido em estágios introduz uma abordagem modular para a concepção de sistemas concorrentes que combinam eventos e *threads*. Nesta abordagem, o processamento de tarefas é dividido em uma série de módulos auto-contidos, batizados de estágios, ligados apenas por filas de eventos.

O conceito de estágios foi proposto pela arquitetura SEDA (do inglês *Staged Event Driven Architecture*) (Welsh2001), originalmente projetada para implementação de servidores altamente concorrentes. Nessa abordagem, uma requisição de usuário é representada por eventos que são processados por uma série de estágios ligados através de filas de eventos. A Figura 2.4 ilustra os componentes básicos do modelo de concorrência orientado a estágios.

Sistemas que utilizam essa abordagem normalmente exploram a característica modular de estágios para sintonizar o ambiente de execução de aplicações, pois eles representam barreiras de controle ao acesso à memória e outros recursos. Assim, as políticas de escalonamento e o compartilhamento de recursos são locais para cada estágio, o que permite que o controle do nível de

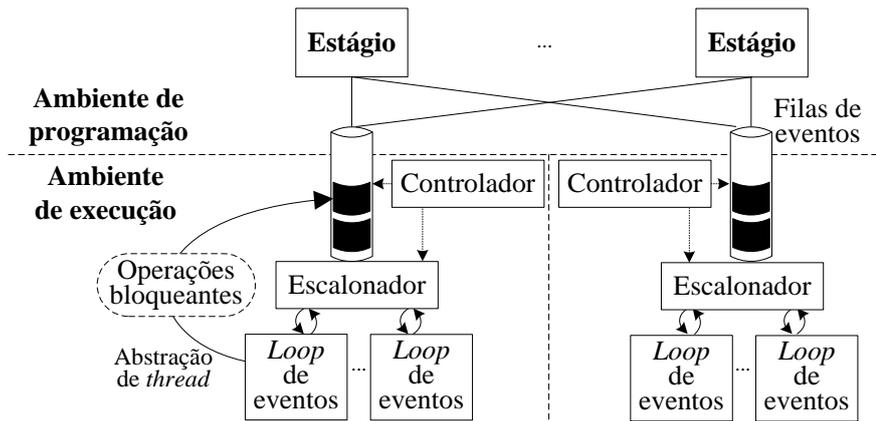


Figura 2.4: Representação do modelo de concorrência híbrido em estágios.

concorrência de sistemas aconteça em uma granularidade mais fina, de forma adequada para as características específicas de cada estágio.

A Figura 2.5 contém uma representação da estrutura interna de um estágio de acordo com a arquitetura SEDA. Estágios atuam como entidades orientadas a eventos independentes, porém são capazes de bloquear internamente (por exemplo, ao chamar uma biblioteca externa ou usando chamadas de sistema bloqueantes), pois utilizam várias *threads* internamente.

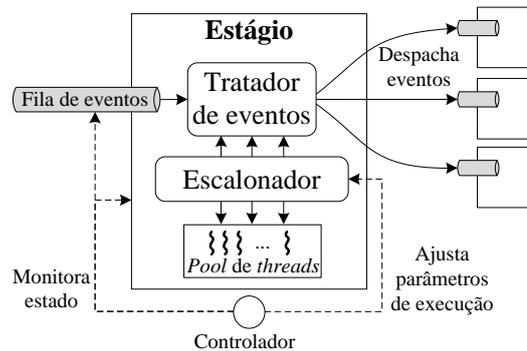


Figura 2.5: Componentes internos de um estágio da arquitetura SEDA.

Cada estágio define um tratador de eventos, que implementa sua funcionalidade, uma fila de eventos e um *pool* de *threads*. Cada *thread* remove eventos da fila de eventos do estágio e executa instâncias concorrentes do tratador de eventos. A execução do tratador de eventos, por sua vez, pode gerar um ou mais eventos que são expedidos para a fila de eventos de outros estágios, e assim, o processamento de uma requisição é realizado na forma de um *pipeline*. O escalonador controla o consumo de eventos para ajustar o nível de concorrência de estágios através de estruturas chamadas controladores, que ajustam dinamicamente os parâmetros de execução do escalonador usando informações tal como o comprimento de fila de eventos, o tempo de resposta do estágio (sua latência de execução), e sua vazão de eventos para outros estágios.

Uma aplicação que segue a arquitetura em estágios pode ser representada conceitualmente como um grafo orientado, onde os nós representam instâncias de estágios e os vértices representam o fluxo de eventos trocados entre eles. Por exemplo, o grafo ilustrado na Figura 2.6 contém a estrutura em estágios de um servidor *web* de páginas estáticas (Welsh2001).

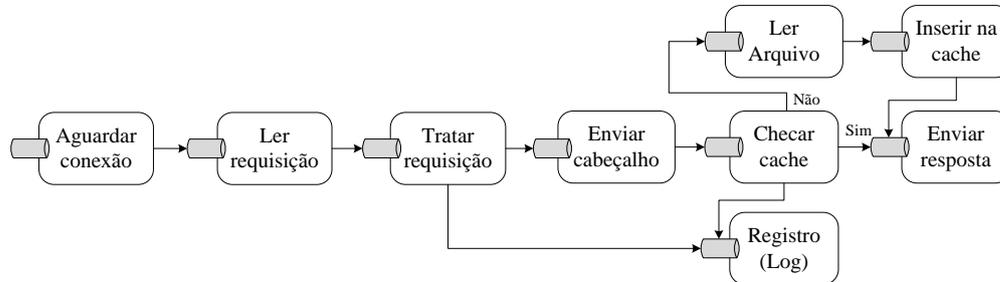


Figura 2.6: Estrutura simplificada de um servidor *web* de páginas estáticas orientado a estágios.

A possibilidade de ajuste fino de concorrência permite calibrar sistemas baseados em estágios e obter melhor desempenho. Cada estágio pode manipular eventos com uma política de escalonamento diferente, de acordo com sua granularidade e padrão de uso de recursos. Welsh *et al.* (Welsh2002) propõe uma série de estruturas chamadas de controladores que podem, por exemplo, monitorar o estado interno de estágios para identificar gargalos indesejados. Com base nessas informações, o programador pode dividir estágios ou combiná-los para melhorar a latência ou vazão de eventos conforme o desejado. Sistemas orientados a eventos e baseados em *threads* oferecem menos flexibilidade nesse aspecto.

Discutiremos a seguir três implementações que utilizam o modelo de concorrência híbrido em estágios: a plataforma Sandstorm (Welsh2001), a arquitetura MEDA (Han2009) e a linguagem de programação concorrente Aspen (Upadhyaya2007).

Sandstorm (Welsh2001) é uma plataforma para auxiliar o desenvolvimento e execução de aplicações baseada nos princípios da arquitetura SEDA, implementada inteiramente em Java. Cada módulo da aplicação, ou estágio, é modelado como uma classe que implementa uma interface simples definida para tratadores de eventos, na qual um método chamado `handleEvents()` é usado para processar lotes de eventos extraídos da fila de entrada do estágio.

A plataforma Sandstorm foi originalmente projetada para a implementação de serviços de internet. Todos os estágios de uma aplicação construída nessa plataforma são agregados em um único espaço de endereçamento e executam dentro de um único processo de sistema operacional. O isolamento de recursos entre estágios é deixado sob responsabilidade exclusiva do progra-

mador. Além disso, as conexões entre diferentes estágios são realizadas por meio de uma fila de eventos em memória compartilhada, tornando-as inadequadas para ambientes com memória distribuída.

Na plataforma Sandstorm, a conectividade entre estágios é determinada exclusivamente durante a execução da aplicação através da instanciação de uma classe que implementa a interface `StageIF`, fazendo-se uso da propriedade de carregamento dinâmico de classes de Java. Um ponto importante com relação ao projeto de Sandstorm diz respeito ao alto grau de acoplamento entre estágios, devido à necessidade de se determinar explicitamente o estágio de destino de um evento durante a especificação do estágio onde o evento se origina. A comunicação entre estágios é precedida pela recuperação do objeto da classe `StageIF` pelo sistema de execução e os eventos são enfileirados diretamente em sua fila de eventos. Essa necessidade dificulta o reuso de estágios em aplicações diferentes.

MEDA (Han2009) é uma arquitetura baseada em SEDA, projetada especificamente para tratar o paradigma mestre-trabalhador, comumente usado em programação paralela e distribuída. MEDA explora o fato de que aplicações que usam esse paradigma são naturalmente estruturados em etapas. O mestre normalmente executa a inicialização, distribuição de trabalho e coleta/com-binação de resultados intermediários. Trabalhadores normalmente processam unidades de trabalho e retornam resultados para o mestre. Na arquitetura MEDA, há estágios pré-definidos correspondentes a essas etapas de processamento. Estágios relacionados aos trabalhadores são replicados ao longo de máquinas da rede.

A arquitetura MEDA estende o modelo de processo único de SEDA para ambientes com memória distribuída. Nessa arquitetura, filas de eventos são implementadas como filas remotas, permitindo que os estágios de uma aplicação executem em nós de rede distintos.

MEDA não trata o controle de concorrência interna de cada estágio, porém seus autores exploram a flexibilidade do escalonamento em nível de usuário para aumentar o rendimento (vazão de eventos) e reduzir o tempo de resposta de estágios específicos através da introdução de prioridades na gestão das filas de eventos.

Aspen (Upadhyaya2007) é uma linguagem de programação concorrente na qual programas são estruturados como um conjunto de módulos, de forma similar à abordagem adotada por SEDA. Módulos Aspen se comunicam através de filas de eventos, trazendo o conceito de estágios para o nível da linguagem de programação. Porém, além de uma fila de eventos de entrada, cada módulo Aspen possui uma ou mais filas de saída. Conexões entre os módulos são

realizadas através da associação das filas de saída a estágios específicos, e são definidas em um arquivo de configuração separado do código da aplicação. Essa característica facilita o reuso de módulos que implementam funções comuns em outras aplicações.

A linguagem Aspen tem como objetivo principal a codificação de serviços de rede. Para isso, introduz o conceito de fluxo, que capta a ideia de elementos de trabalho que devem ser processados sequencialmente e em ordem. Aspen atribui todos os elementos de trabalho relacionados ao mesmo fluxo a uma única *thread*, garantindo sua sequencialidade.

Cada módulo Aspen tem seu espaço exclusivo de endereçamento: variáveis e descritores de arquivos são privados para cada módulo. Isso evita problemas de compartilhamento de memória e recursos, permitindo que o modelo da linguagem Aspen possa ser implementado em ambientes com memória distribuída.

Internamente a cada módulo, Aspen faz algumas provisões para evitar condições de corrida. Em primeiro lugar, o programador pode especificar variáveis especiais do tipo *per-flow*. Aspen cria automaticamente uma cópia independente de tais variáveis para cada fluxo. Isso facilita a manutenção do estado entre os elementos de trabalho em um único fluxo, permitindo o uso da memória global de um módulo de uma forma protegida. Em segundo lugar, a linguagem Aspen não permite a execução paralela de fluxos em que haja acesso compartilhado a recursos como memória ou descritores de arquivos.

2.2 Discussão

A seção anterior propôs uma classificação de modelos de concorrência híbrida baseada no viés que cada tipo de modelo apresenta em relação às abstrações de concorrência.

A abstração baseada em estágios não apresenta um viés claro para *threads* ou eventos, buscando equilibrar os benefícios oferecidos por cada uma dessas abstrações. Assim, o modelo de estágios permite que programadores escolham o modelo de concorrência mais conveniente para descrever aplicações concorrentes de forma flexível.

Discutiremos a seguir quatro questões relevantes referentes ao uso do modelo em estágios no desenvolvimento de aplicações concorrentes.

Desenvolvimento Modular, Desacoplado e em Etapas

Tanto a concorrência híbrida baseada em *threads* e a orientada a estágios são apropriadas para descrever fluxos de controle grandes, nos quais a especificação de tarefas se dá em granularidade grossa. A abordagem baseada em *threads* divide fluxos de controle em pedaços de forma transparente, tipicamente delimitados por chamadas de sistema bloqueantes. Similarmente, a abordagem baseada em estágios propõe decompor fluxos de controle em uma série de passos, porém, através de uma abordagem modular na qual a responsabilidade de dividir o fluxo de controle em estágios é deixada ao programador.

A escolha do modelo de concorrência adequado depende fortemente das métricas de interesse. A abordagem orientada a estágios oferece maior flexibilidade em termos de escolha da abstração de concorrência, pois ela expõe *threads* e eventos ao programador. A escolha do nível de granularidade de estágios permite controlar o modelo de programação em partes específicas de uma aplicação (Gribble2001).

Filas de eventos podem resultar em grande aumento de latência quando sobrecarregadas, o que dificulta seu uso em cenários de aplicações que possuem restrições de latência, como em sistemas de tempo real. Em tais cenários, a divisão da aplicação em passos com granularidade mais grossa e síncronos podem ser mais apropriada pois reduz o uso de filas de eventos. Essa abordagem favorece o uso de modelos de concorrência híbrida baseada em *threads*. Em contrapartida, cenários na qual a latência não é um requisito decisivo podem se beneficiar da divisão de tarefas com granularidade mais fina. Por exemplo, a vazão de processamento é um fator prioritário com relação a latência de atendimento a clientes individuais em aplicações de processamento de imagem (Gordon2010) ou em servidores para transmissão de vídeos sob demanda (Upadhyaya2007).

A Figura 2.7 ilustra três possibilidades de se estruturar uma aplicação com diferentes granularidades que usam estratégias distintas de concorrência.

Em (a), o uso de granularidade fina, onde cada passo sequencial de processamento da aplicação é executado em um estágio, reflete um ambiente de execução primariamente orientado a eventos. Em (b), dois passos sequenciais de processamento são agrupados em um só estágio, eliminando a comunicação via fila de eventos enquanto o restante da estrutura da aplicação permanece inalterada. Em (c), o agrupamento de todos os passos sequenciais da aplicação em um único estágio irá se refletir em uma estratégia de concorrência puramente baseada em *threads*, onde cada fluxo de processamento concorrente ocorre inteiramente em uma *thread* exclusiva.

O mecanismo de comunicação em SEDA possui um alto grau de acopla-

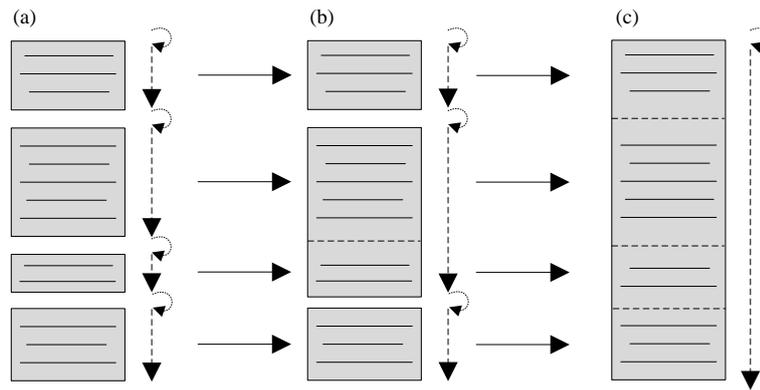


Figura 2.7: A mesma especificação de uma aplicação em estágios com diferentes granularidades.

mento entre estágios, devido à necessidade de se determinar explicitamente o estágio de destino durante a especificação do estágio onde o evento se origina. Essa necessidade quebra o encapsulamento de estágios e dificulta o reuso da mesma especificação em outras aplicações. Outro problema com esta abordagem é que o fluxo de eventos entre os estágios é definido de forma implícita durante a especificação dos tratadores de eventos, dificultando a visualização do grafo de relacionamento de estágios que compõem a aplicação.

O uso de abstrações de comunicação em alto nível, como as filas internas de saída de módulos Aspen, permite o reuso de estágios que implementem funcionalidades semelhantes, além de explicitar o fluxo de eventos da aplicação e o relacionamento entre estágios.

Uso de Threads em Nível de Usuário

Modelos de concorrência híbridos tentam combinar a expressividade de *threads* com a flexibilidade de sistemas orientados a eventos, expondo, em diferentes graus, o controle do escalonador para o programador. O uso de escalonamento e abstrações de *thread* em nível de usuário parece ser um mecanismo importante para atingir flexibilidade e controle. As abordagens orientadas a estágios identificadas neste trabalho usam diretamente *threads* de sistema operacional dentro do *pool* de estágios e poderiam se beneficiar do uso dessas abstrações em nível de usuário (Upadhyaya2007).

Threads em nível de usuário são flexíveis em muitos sentidos. Elas permitem a separação entre aplicações e *threads* de sistema operacional, permitindo inovações em ambos os lados (vonBehren2003b), como, por exemplo, o uso de políticas de escalonamento em nível de usuário específicas, que não são limitadas pelo algoritmo de escalonamento do sistema operacional.

As sobrecargas de sincronização e troca de contexto de *threads* podem ser reduzidas pelo uso de *threads* em nível de usuário mesmo em ambientes multiprocessados, pois não usam recursos de *kernel*. Assim, elas podem ser dimensionadas em maior quantidade com relação às *threads* de sistema operacional.

Isolamento de Estado Entre Tarefas Concorrentes

O modelo híbrido em estágios reforça o isolamento de recursos entre estágios distintos através da presença de filas de eventos. Recursos como ponteiros de memória e descritores de arquivos devem ser cuidadosamente gerenciados para garantir que não haja compartilhamento simultâneo de dados em partes diferentes da aplicação. A abstração oferecida para especificação de estágios procura esconder os aspectos do gerenciamento de concorrência do desenvolvedor ao controlar criação de *threads* e gerir o ambiente de execução da aplicação, no entanto, a presença de concorrência dentro de estágios introduz novamente o problema de condições de corridas para tratadores de eventos que têm acesso a uma memória compartilhada dentro de estágios e requer sincronização explícita pelo programador.

Abordagens existentes para concorrência em estágios não têm, em geral, prestado muita atenção para a questão da concorrência interna. A exceção é o modelo da linguagem Aspen, que força a proteção de dados globais, porém o protótipo inicial do sistema não inclui este recurso.

Múltiplas instâncias de um mesmo tratador de eventos com forte isolamento de estado são usadas em modelos de concorrência que executam sobre linguagens funcionais (Upadhyaya2007, Armstrong1996) para impedir o compartilhamento de recursos entre os segmentos de trabalho concorrentes. Técnicas de isolamento em máquinas virtuais e *sandboxing* também podem ser usadas sobre linguagens de *script* (Ururahy2002). Nesses casos, a troca de dados entre tarefas acontece exclusivamente via passagem de mensagens, favorecendo um ambiente de execução livre de sincronização.

Execução de Aplicações em Ambientes Distribuídos

A atual popularidade de ambientes de computação elástica e virtualização enfatizam o valor de modelos que permitem a execução de aplicações em ambientes onde não há compartilhamento de recursos (ou ambientes *shared-nothing*). Uma alternativa para a adaptação de sistemas híbridos para arquiteturas de memória distribuída é replicar o ambiente de execução através de

processos executando em múltiplos *hosts* e equilibrar a carga de trabalho entre os processos disponíveis. Não apenas servidores mas também aplicações que envolvem computação intensiva podem se beneficiar de modelos de concorrência híbridos.

Ambientes em *clusters*, onde cada nó oferece um ambiente de execução com muitos núcleos e protocolos de redes locais cada vez mais rápidos motivam a necessidade de ferramentas capazes de executar partes diferentes de uma aplicação em ambientes desacoplados. Métodos de passagem de mensagens, como a biblioteca MPI (Pacheco1997), são usualmente aplicados para executar aplicações em tais ambientes. O modelo MapReduce (Ghemawat2004) para o processamento de grandes massas de dados em ambientes *shared-nothing* é outro exemplo de padrão que se beneficia da execução desacoplada de aplicações.

Com exceção da arquitetura MEDA, as abordagens que descrevemos não oferecem soluções para os problemas de compartilhamento de recursos e de comunicação entre tarefas concorrentes executando em ambientes com memória distribuída. Lidar com essas questões diretamente na infra-estrutura de concorrência facilita a implementação de aplicações que podem se beneficiar da execução em ambientes distribuídos.

3

Uma Extensão do Modelo de Concorrência em Estágios

Este capítulo descreve uma extensão do modelo de programação orientado a estágios que trata as questões discutidas na seção 2.2. Nossa hipótese é que a flexibilidade proporcionada por estágios pode ser estendida através de um ambiente de execução cooperativo e isolado para dissociar a especificação da lógica de funcionamento das aplicações das decisões de programação relacionadas ao ambiente de execução.

3.1

Modelo de Programação

O objetivo principal do modelo proposto neste trabalho é incentivar o desacoplamento entre a especificação de estágios da aplicação e o mapeamento de partes da aplicação em unidades de execução individuais.

Para isso, o modelo de programação se apoia parcialmente nos conceitos de desenvolvimento em etapas da metodologia PCAM, proposta em (Foster1995) para modelar sistemas paralelos. A adoção desse tipo de modelagem procura incentivar o desenvolvimento de aplicações em estágios escaláveis e reutilizáveis, postergando-se considerações dependentes do ambiente de execução até os últimos momentos possíveis. A Figura 3.1 contém uma representação gráfica das etapas do processo de desenvolvimento PCAM.

Pela metodologia PCAM, o desenvolvimento de uma aplicação se dá em quatro etapas distintas: particionamento, comunicação, aglomeração e mapeamento. A fase de *particionamento* expõe as oportunidades para execução paralela, decompondo o problema em tarefas independentes com a granularidade mais fina possível. Na etapa de *comunicação*, o fluxo de informações entre as tarefas é especificado. A terceira fase, a *aglomeração*, já é mais próxima ao ambiente no qual a aplicação executará. Nessa etapa, as tarefas identificadas na fase de partição são aglomeradas de modo a obter-se um nível razoável de granularidade de tarefas paralelas. A quarta e última fase da metodologia PCAM é a etapa de *mapeamento*, na qual as tarefas aglomeradas são mapeadas para processadores físicos.

Em nosso modelo de programação, o projeto de uma aplicação começa

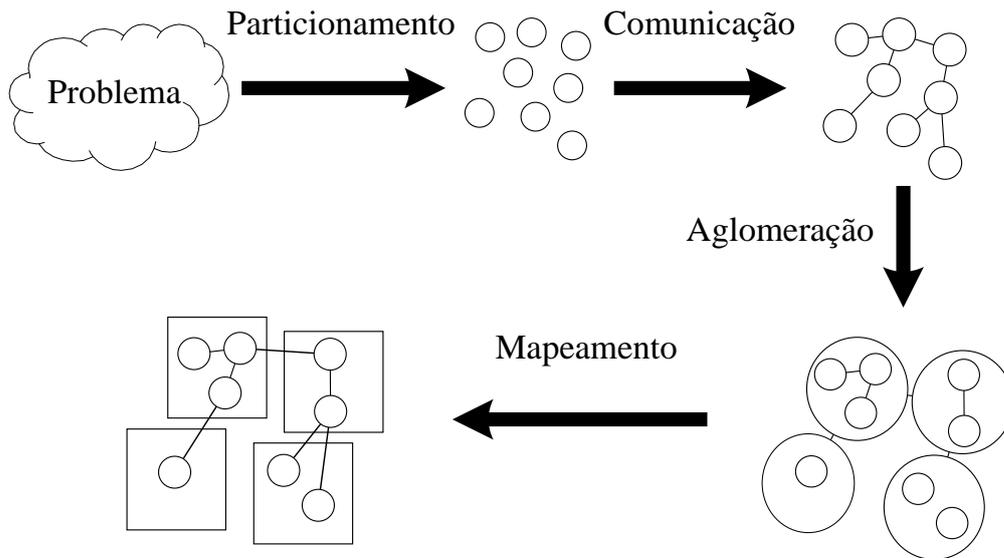


Figura 3.1: Etapas do processo de desenvolvimento PCAM.

pela sua decomposição em um conjunto de estágios. Essa decomposição é análoga à etapa de particionamento do processo PCAM e evidencia oportunidades de concorrência, produzindo o maior número possível de tarefas independentes.

A divisão de tarefas no processo PCAM baseia-se ou em decomposição de domínio ou em decomposição funcional. A decomposição em domínio leva em consideração a divisão de dados a serem processados concorrentemente enquanto o foco da decomposição funcional é a divisão da aplicação em tarefas concorrentes. No modelo de estágios, a aplicação é particionada em unidades funcionais, ou estágios, que definem um tratador de eventos e que podem, por sua vez, gerar outros eventos a serem tratados por outros estágios. Assim, o particionamento nesta etapa é sempre baseado em funcionalidade. Porém, diversas instâncias concorrentes de um mesmo estágio podem executar paralelamente, processando dados distintos durante sua execução, provendo-se o particionamento em domínio.

Diferentemente do que ocorre em SEDA, estágios em nosso modelo desconhecem os estágios que irão consumir os eventos emitidos por eles. Em vez de emitir eventos para estágios consumidores específicos, os eventos são enviados para *portas de saída* (Papadopoulos1998), ligadas, em uma etapa posterior, a canais unidirecionais assíncronos chamados de *conectores*. Assim, nosso modelo facilita a reutilização de estágios em diferentes aplicações.

Na etapa de comunicação, a estrutura de estágios da aplicação é definida pela ligação de portas de saída a filas de eventos de outros estágios através de conectores. A ligação de estágios é similar à ligação de componentes em linguagens de descrição de arquiteturas (ADLs) (Garlan2010). O resultado

dessa etapa é um grafo direcionado, onde nós representam estágios de processamento e as arestas representam conectores. Nessa etapa, como em PCAM, o programador pode verificar os padrões de comunicação resultantes. A Figura 3.2 mostra um exemplo de um grafo com quatro estágios (*A*, *B*, *C* e *D*), suas portas de saída, e quatro conectores.

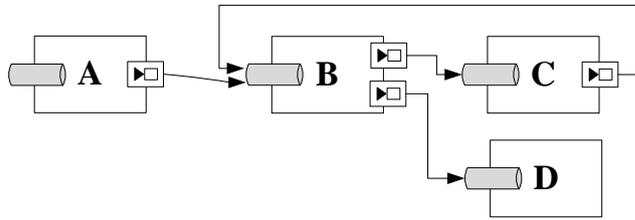


Figura 3.2: Representação gráfica de uma aplicação contendo quatro estágios, suas portas e seus conectores.

Na terceira etapa de desenvolvimento, análoga à etapa de aglomeração do processo PCAM, os estágios do grafo da aplicação são agrupados em *aglomerados*, que serão mapeados individualmente para unidades de execução independentes. Aglomerados representam a granularidade na qual diferentes políticas de escalonamento serão aplicadas durante a execução da aplicação, e a sua dimensão pode variar de um único estágio à toda a aplicação. A Figura 3.3 ilustra uma possível divisão do gráfico da Figura 3.2, onde os estágios *B* e *C* são agrupados em um mesmo aglomerado. Uma vez que o processo de aglomeração é totalmente separado das etapas anteriores no projeto da aplicação, o desenvolvedor é livre para experimentar com diferentes configurações de aglomeração de estágios a fim de alcançar um bom equilíbrio entre políticas de escalonamento em partes específicas da aplicação e a granularidade de computação com relação a comunicação.

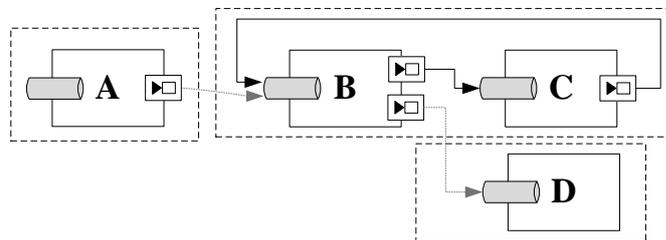


Figura 3.3: Uma possível configuração de aglomeração do grafo da Figura 3.2.

A quarta e última etapa, o mapeamento, ocorre no momento da execução da aplicação. Nessa etapa, cada aglomerado definido no grafo de estágios da aplicação é mapeado para um processo de sistema operacional que executa em uma máquina arbitrária. O fato desse mapeamento ser adiado para a última

etapa permite que o desenvolvedor execute o mesmo gráfico de estágios em diferentes configurações de aglomeração e mapeamento.

Nas seções a seguir, discutiremos com mais detalhes cada um dos elementos de nosso modelo de programação orientado a estágios.

3.2 Estágios

A primeira etapa de desenvolvimento consiste no particionamento de uma aplicação em estágios que se comunicam através da troca de eventos. Durante a execução da aplicação, instâncias concorrentes do tratador de eventos são criadas para prover o paralelismo de domínio em estágios. Instâncias são executadas por *threads* do *pool* para processar os dados consumidos da fila de eventos.

Em SEDA, um único estado global é compartilhado entre as instâncias de seu tratador de eventos. Isso obriga que estágios sincronizem o acesso e a alteração de dados. O uso de gerenciamento cooperativo de tarefas reduz a sobrecarga de troca de contexto e de sincronização entre tarefas concorrentes. Porém, a extensão desse tipo de controle de tarefas em ambientes com múltiplas *threads* de sistema operacional significa trazer de volta à tona os problemas relacionados a sincronização de acesso a recursos compartilhados para impedir condições de corrida.

Entendemos que a chave para se aplicar o gerenciamento de tarefas cooperativo em ambientes multiprocessados esteja no controle rigoroso do compartilhamento de recursos, impedindo que instâncias concorrentes de tratadores de eventos de um estágio atuem simultaneamente sobre os mesmos dados. Assim, cada instância em execução tem o seu próprio estado e toda a comunicação é realizada através de passagem de mensagens.

A estrutura interna de execução de um estágio n é composta por três partes:

1. Uma fila de eventos q_n como ponto de entrada de eventos para o estágio;
2. Um conjunto I_n de instâncias concorrentes que mantêm estado próprio;
3. Um conjunto P_n de portas como interface de saída de eventos.

A Figura 3.4 contém uma representação gráfica dos componentes de um estágio.

O conjunto de portas de saída P_n é visível por todas as instâncias concorrentes de um estágio n . O ambiente de execução de instâncias oferece uma primitiva para permitir o despacho de eventos para uma porta ($p \in P_n$)

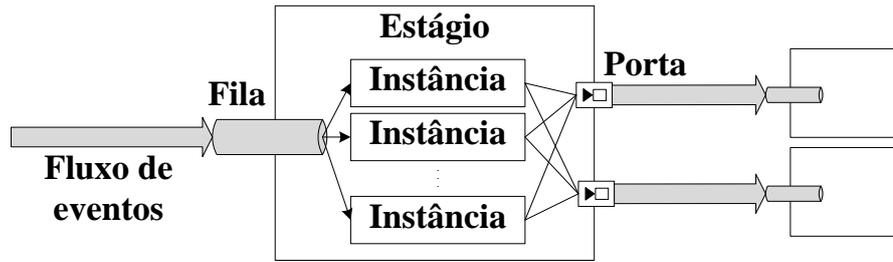


Figura 3.4: Representação gráfica de um estágio em execução com duas portas de saída.

e o agregado de eventos emitidos por todas as instâncias I_n compõem o fluxo sequencial de saída de eventos resultante na porta p .

Tratadores de eventos são funções capazes de processar eventos retirados da fila de entrada e, como atuam dentro do estado interno de uma instância, não precisam lidar com condições de corrida. A execução de um tratador pode, por sua vez, resultar no enfileiramento de novos eventos para as portas de saída do estágio.

Nosso modelo admite dois tipos de instâncias: transientes e persistentes. Em instâncias transientes, as alterações no estado interno são descartadas ao fim de cada execução, assim, o estado interno de instâncias transientes antes de sua execução é sempre o mesmo. As instâncias persistentes mantêm seu estado interno entre invocações do tratador de eventos. Assim, instâncias persistentes podem ser usadas, por exemplo, para manter um contador interno, ou para acumular resultados anteriores.

Em estágios com instâncias persistentes, o consumo de eventos é serializado. Isso permite que eles funcionem como máquinas de estado sequenciais. O paralelismo de execução dentro de estágios só é possível através da criação de um número arbitrário de instâncias transientes pelo ambiente de execução.

3.3 Comunicação

Os estágios definidos na etapa anterior, em geral, não executam de forma independente. O processamento a ser realizado em um estágio normalmente requer dados que são gerados por outros estágios. Fluxos de eventos devem ser transferidos entre estágios de modo a permitir que o processamento ocorra como um *pipeline*. Esse fluxo de eventos é especificado na etapa de comunicação do nosso modelo.

O resultado da etapa de comunicação é um grafo que representa a estrutura da aplicação. O grafo da aplicação é composto por um conjunto de estágios em que eventos fluem de suas portas de saída para a fila de entrada

do próximo estágio. O envio de eventos através de canais de comunicação envolve um custo adicional e a ligação explícita dessas interfaces nos ajuda a pensar quantitativamente sobre questões de localidade de processamento e os custos de comunicação envolvidos. Estágios possuem um único ponto de entrada (sua fila de eventos) e um número arbitrário de interfaces de saída de eventos, permitindo que eles sejam vinculados e compostos livremente através da ligação de suas interfaces através de estruturas chamadas de *conectores*.

Uma aplicação pode ser definida como um grafo $G = (S, C)$, onde S é um conjunto de estágios e C é um conjunto de conectores que ligam uma porta de saída p de um estágio i a uma fila de eventos de entrada de um estágio j , nos quais $C \subseteq (p, j)$, com $(p \in P_i)$ e $(i, j \in S)$. Cada porta de saída de um estágio pode estar ligada no máximo a um conector, enquanto uma fila de eventos de entrada de um estágio pode ter um número arbitrário de conectores incidentes.

A Figura 3.5 contém uma representação gráfica de três grafos de estágios que usam padrões distintos para processamento paralelo de eventos.

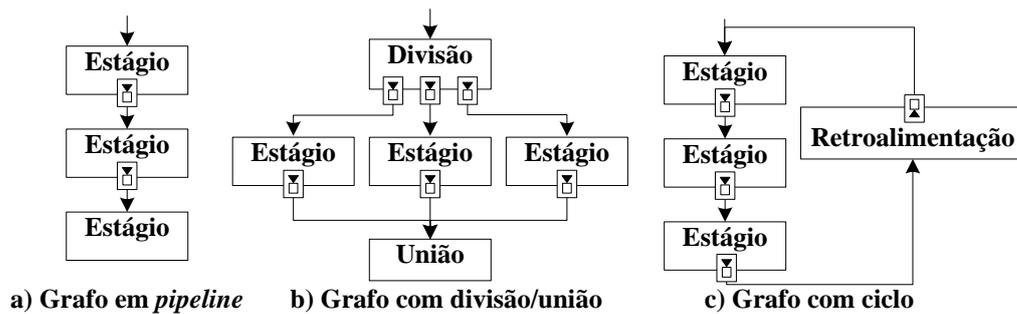


Figura 3.5: Grafos de estágios estruturados em três padrões de comunicação diferentes.

O grafo apresentado na parte (a) da Figura 3.5 corresponde a um *pipeline* simples. O grafo apresentado na parte (b) representa um *pipeline* com ramificação, contendo um estágio para divisão de eventos com três portas de saída e um estágio para união de fluxos de eventos. O grafo apresentado na parte (c) contém um ciclo devido à presença de um estágio de retroalimentação.

Os conectores de um grafo de estágios representam canais independentes para transporte de dados na aplicação. Esse transporte de dados é realizado através de uma interface de comunicação baseada em troca de mensagens. Mensagens são entregues às filas de eventos dos estágios de destino de acordo com os conectores correspondentes do grafo e processadas assincronamente.

Uma vez que ligação entre estágios é estabelecida separadamente de suas especificações, o grafo de uma aplicação pode ser manipulado livremente antes de sua execução. Por exemplo, a Figura 3.6 contém um exemplo de aplicação representada por dois grafos equivalentes, porém com configurações diferentes.

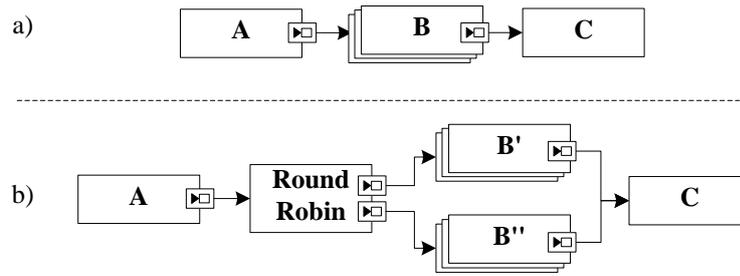


Figura 3.6: Exemplo de grafos equivalentes para uma aplicação.

O grafo da parte (a) da Figura 3.6 é um *pipeline* simples contendo um estágio *B* com instâncias transientes. O grafo da parte (b) é equivalente, porém com a adição do estágio *Round Robin*, que divide os eventos recebidos entre duas réplicas exatas do estágio *B* de forma alternada. Uma vez que as instâncias do estágio *B* não possuem estado persistente, os dois grafos representam aplicações equivalentes. Nesse caso, a configuração do grafo é modificada, porém os códigos dos tratadores de evento dos estágios *A*, *B* e *C* permanecem inalterados.

Essa opção é um meio termo entre a visão seguida por SEDA, que usa a construção do grafo em tempo de execução (o destino dos eventos emitidos por estágios pode ser determinado dinamicamente), e a seguida por linguagens estáticas como Aspen (Upadhyaya2007), que usa a construção do grafo em tempo de compilação. Essa abordagem permite a união da flexibilidade da construção dinâmica do grafo de estágios com a possibilidade da verificação de propriedades da aplicação antes de sua execução, por exemplo, para garantir que as portas de saída dos estágios estejam devidamente conectadas.

3.4

Aglomerção de Estágios

O foco das duas primeiras etapas de nosso modelo é definir a semântica de funcionamento dos estágios e a estrutura geral do grafo da aplicação. Nessas etapas, poucas suposições sobre o ambiente de execução da aplicação são consideradas. Na etapa de aglomeração, o programador leva em consideração propriedades específicas do ambiente de execução, além das características individuais dos estágios, para definir um conjunto de aglomerados de estágios que formam uma partição do grafo da aplicação. Aglomerados representam domínios de escalonamento independentes e têm o objetivo de promover a política de divisão de recursos disponíveis para os estágios. Assim, nessa etapa, o programador deve considerar três fatores: o domínio das decisões de escalonamento para partes específicas da aplicação, a complexidade de processamento dos estágios e os custos de comunicação envolvidos.

Em SEDA, estágios constituem unidades de execução independentes que possuem um escalonador e um *pool* de *threads* de sistema operacional próprios. Essa característica permite o uso de operações bloqueantes em tratadores de evento, porém, introduz sobrecargas de gerenciamento de tarefas e *threads* proporcionais ao número de estágios da aplicação. Na implementação de SEDA, os controladores de cada estágio operam de forma independente, ajustando o tamanho do *pool* de *threads* de acordo com as condições de carga percebidas, o que pode causar problemas em estágios interdependentes (Gordon2010). Por exemplo, provocando uma oscilação do número de *threads* pelo fato de que cada estágio pode alterar o *pool* continuamente em resposta ao comportamento de outro estágio.

Trabalhos posteriores tratam esse problema através do uso de controladores globais, que levam em consideração o estado interno de toda a aplicação para ajustar os parâmetros de execução de estágios e aplicar um limite global no número de *threads* alocadas para o sistema (Li2006), ou permitir que estágios compartilhem um *pool* de *threads* único (Gordon2010). Essas técnicas requerem que estágios executem em um ambiente compartilhado e podem ser aplicada de forma independente em cada aglomerado definido. O escalonamento de tarefas em ambientes orientados a estágios é um complexo problema na sua forma geral, e ainda constitui um campo de investigação ativo na academia (Bharti2005, Hakeem2010, Gordon2010).

Agrupar o processamento de estágios em domínios de escalonamento distintos pode oferecer diversos benefícios. Primeiro, aglomerados podem ser usados para isolar estágios que fazem uso intensivo de recursos em um ambiente de execução restrito, impedindo que estes monopolizem todos os recursos disponíveis. Segundo, aglomerados distintos podem definir políticas de alocação de recursos adaptadas para as necessidades específicas de partes da aplicação em questão. Por exemplo, fazer uso de uma política que aloca uma *thread* por CPU disponível pode ser a melhor opção para alcançar maior desempenho em um aglomerado composto essencialmente de estágios que não bloqueiam. Porém, partes de uma aplicação que necessitam de operações de entrada e saída bloqueantes podem de beneficiar de um ambiente de execução com um número maior de *threads* em relação a quantidade de processadores. Finalmente, estágios agrupados em um mesmo aglomerado devem ser mapeados em um mesmo ambiente de execução, permitindo que estes troquem recursos locais, enquanto aglomerados diferentes executam em ambientes sem memória compartilhada. Assim, o modelo permite que diferentes partes da aplicação agreguem recursos físicos distribuídos no ambiente de execução para atingir um maior nível de paralelismo.

O desempenho de uma aplicação em estágios depende de diversos fatores: o compromisso entre o número de *threads* e processadores disponíveis, o número de instâncias concorrentes a executar, a granularidade de processamento dos estágios e a sobrecarga de comunicação introduzida. A aglomeração de tarefas é especialmente importante para estágios que trocam grandes quantidades de dados. Estágios agrupados em um mesmo aglomerado podem se comunicar através de filas de eventos em memória compartilhada, permitindo o uso de otimizações de comunicação, como a utilização direta de *buffers* em memória. O custo relativo de comunicação de um estágio pode ser definido como a razão entre o tempo de computação T (ou latência de processamento de um estágio) e a latência de comunicação C . Estágios com granularidade fina possuem uma razão $\frac{T}{C}$ pequena, enquanto estágios com granularidade grossa correspondem a uma razão $\frac{T}{C}$ maior.

O agrupamento de um conjunto de estágios em uma única unidade de execução permite que algoritmos de escalonamento façam a atribuição de instâncias de estágios a *threads* de sistema operacional de forma a limitar o domínio de escalonamento para partes específicas do grafo de estágios e a equilibrar a relação entre processamento e comunicação de estágios.

3.5 Mapeamento de Aglomerados

A última etapa de nosso modelo consiste no mapeamento dos aglomerados em unidades de execução exclusivas, e ocorre no momento da execução da aplicação. O mapeamento determina a localização onde cada aglomerado do grafo irá executar, e estágios da aplicação agrupados em um aglomerado devem obrigatoriamente ser mapeados a uma mesma unidade de execução, que é um processo de sistema operacional.

Processos executam instâncias de estágios de forma cooperativa através de um *pool* de *threads* compartilhado. Cada processo é composto por um escalonador de tarefas, um *pool* de *threads* e um controlador para ajuste dinâmico dos parâmetros de execução de forma similar a um estágio da arquitetura SEDA. Porém, nesse caso, o controlador tem acesso ao estado interno de todos os estágios mapeados ao processo.

O mapeamento de aglomerados para unidades de execução determina a distribuição de tarefas entre os processos disponíveis e o método de comunicação a ser utilizado em conectores. Conectores que ligam estágios mapeados a um mesmo processo utiliza uma interface de comunicação local em memória compartilhada, como em SEDA, e estágios mapeados a processos diferentes se comunicam através de interfaces de comunicação remota.

Devido ao seu isolamento, processos são usados para prover um mecanismo de proteção e evitar falhas globais, ou prover segurança através da separação do uso de recursos dentro da aplicação, mesmo que executem em um mesmo nó de processamento.

Um conjunto de nós de rede em que cada processo participante contribui com uma quantidade modesta de recursos computacionais, de comunicação e de armazenamento permite um alto grau de multiplexação e compartilhamento de recursos, possibilitando que picos de carga sejam tratados de forma escalável. Além disso, o uso de processos distribuídos também ajuda a melhorar a tolerância a falhas em aplicações.

A Figura 3.7 contém uma representação gráfica do mapeamento de três aglomerados em três unidades de execução distintas.

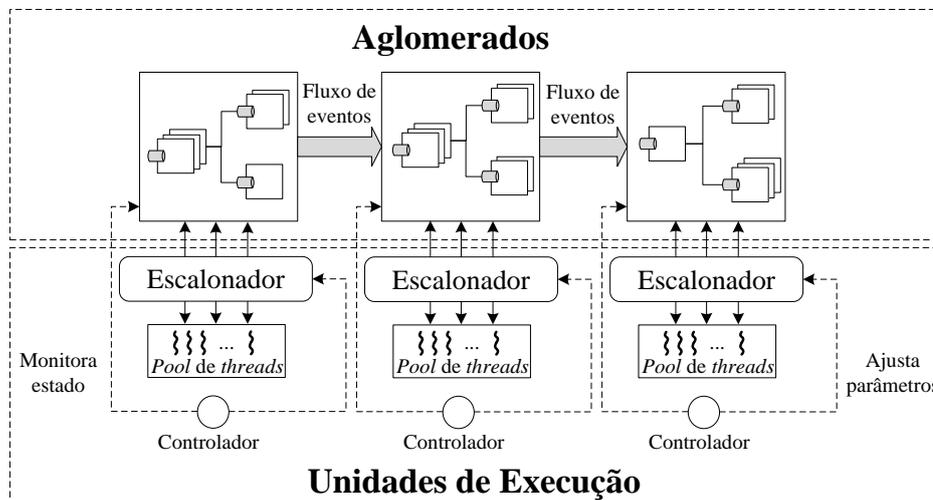


Figura 3.7: Mapeamento de aglomerados de estágios em unidades de execução independentes.

Como o processo de aglomeração e de mapeamento é adiado para as últimas etapas do processo de execução do grafo, uma mesma aplicação pode ser mapeada em diferentes configurações em diversos cenários de distribuição. Assim, o modelo promove uma abordagem exploratória, onde o programador pode experimentar diversas configurações para determinar a que reflete melhores resultados para a aplicação e o ambiente de execução em questão.

A decisão de mapeamento é dependente tanto de características inerentes da aplicação quanto de características específicas de cada ambiente de execução. Por exemplo, a decisão do local de mapeamento de um aglomerado pode ser influenciada pela localização física dos recursos necessários pelos estágios, a quantidade de CPUs e a velocidade relativa com relação a outros processadores do conjunto ou a velocidade da rede que interliga os ambientes de execução. Muitos serviços de processamento de fluxo de eventos que ope-

ram com massas de dados grandes, como previsão do tempo, processamento de *logs* e análise do mercado de ações, inerentemente processam e compõem dados a partir de diferentes domínios autônomos. Distribuição é um requisito fundamental nestas situações.

Um processo que está sobrecarregado com muitas tarefas concorrentes pode ser identificado e a sua configuração de aglomerados pode ser alterada para se obter uma divisão mais equilibrada de acordo com o ambiente de execução e o padrão de troca de dados entre estágios da aplicação. Por exemplo, uma aplicação pode aplicar a técnica ilustrada na Figura 3.6 para agregar o processamento de dois ambientes de execução distintos.

3.6 Discussão

Nesse capítulo nós apresentamos uma extensão para o modelo de eventos em estágios que promove a dissociação entre a especificação da lógica de aplicações e a sua estrutura de execução, estabelecendo um procedimento em etapas para o desenvolvimento de aplicações concorrentes.

Nesse modelo, estágios são inicialmente definidos puramente por seu papel na lógica de funcionamento da aplicação, sem nenhuma preocupação com a localidade de execução, e são unidos por meio de canais de comunicação assíncronos, chamados de conectores, para formar um grafo direcionado que representa o fluxo de eventos dentro da aplicação. Posteriormente, o grafo resultante é particionado em aglomerados, que devem ser mapeados para processos de sistema operacional e podem executar em uma única máquina ou em uma configuração distribuída arbitrária.

Como as etapas de aglomeração e mapeamento são desacopladas da definição do grafo, diferentes partições podem ser configuradas para a mesma aplicação, permitindo que o programador experimente diferentes configurações para equilibrar a utilização de recursos disponíveis no ambiente de execução e a granularidade de computação e comunicação dentro da aplicação.

No modelo SEDA, é possível controlar o número de *threads* dedicadas a cada um dos estágios. Em nossa extensão, os aglomerados permitem que a granularidade do domínio das decisões de escalonamento seja definida pelo programador, dissociando o *pool* de *threads* da lógica de execução dos estágios da aplicação. Além disso, a separação de instâncias de estágios das *threads* de sistema operacional permite um maior nível de experimentação e ajuste.

Essas características são especialmente importantes quando uma única aplicação destina-se à execução em diferentes plataformas: o uso de *hardwares* diferentes e de configurações específicas do sistema operacional implicam

em diferentes parâmetros de configuração da aplicação para obter melhor desempenho de execução.

4

Leda: Uma Implementação do Modelo de Estágios em Lua

Este capítulo descreve uma visão geral da implementação da biblioteca Leda, uma plataforma para execução de aplicações distribuídas com as características do modelo de estágios discutidas no capítulo 3.

Utilizamos a linguagem de programação Lua (Ierusalimschy2006, Ierusalimschy2011) como base para a implementação do modelo proposto. Lua é uma linguagem dinâmica e interpretada por uma máquina virtual leve que oferece recursos úteis para o desenvolvedor de aplicações concorrentes. Especificamente, nós destacamos os mecanismos de funções como valores de primeira classe, escopo léxico e concorrência cooperativa através de co-rotinas (deMoura2009).

Com exceção da primeira etapa do processo de desenvolvimento de aplicações, onde a programação dos tratadores de eventos de estágios é imperativa, as etapas de comunicação, aglomeração e mapeamento são tarefas típicas de configuração. Portanto, a escolha de uma linguagem de *scripting* como Lua foi uma escolha natural para descrever tais etapas. Além disso, como Lua é projetado para ter o seu código chamado a partir de um programa hospedeiro C, é fácil manipular o seu ambiente de execução de forma a satisfazer os requisitos estabelecidos pelo modelo de concorrência proposto. Finalmente, a interoperabilidade de Lua e C é adequada para a codificação dos tratadores de eventos que necessitam lidar com computação intensiva com código compilado quando necessário.

A próxima seção apresenta o modelo de programação de aplicações oferecido pela biblioteca Leda para a especificação de estágios e para a sua composição em um grafo. Em seguida, a Seção 4.2 descreve as características relacionadas à comunicação entre estágios, que podem ser mapeados para um mesmo ambiente de execução ou para ambientes de execução diferentes. Finalmente, a Seção 4.3 apresenta a implementação dos processos que executam os aglomerados que compõem a aplicação em estágios.

4.1

Ambiente de Programação

Nesta seção, descrevemos uma visão geral da biblioteca Leda, que é responsável por prover as operações básicas necessárias em cada etapa de desenvolvimento de aplicações em estágios seguindo o modelo de programação proposto.

4.1.1

Abstrações de Programação Leda

Seguindo o modelo definido no capítulo 3, Leda dá suporte a uma abordagem iterativa para o desenvolvimento de aplicações concorrentes.

Uma visão incremental na qual cada etapa oferece abstrações de programação específicas foi definida para a implementação de aplicações em estágios. Assim, definimos quatro etapas que são ilustradas na Figura 4.1 e descritas a seguir:

1. Etapa de programação – a primeira etapa oferece a abstração de mais alto nível do modelo e consiste na especificação imperativa dos tratadores de eventos de estágios e suas interfaces de saída de eventos através de uma API simples. Nessa etapa, o desenvolvedor trabalha apenas com a abstração *Stage*, que representa estágios individuais, sem a necessidade de se preocupar com a estrutura geral da aplicação.
2. Etapa de comunicação – A segunda etapa define os mecanismos de comunicação entre estágios da aplicação. Nessa etapa, o desenvolvedor trabalha com a abstração *Connector* para a formação de um grafo de estágios de maneira declarativa. Restrições de comunicação entre estágios, como a necessidade de troca de recursos locais, devem ser descritas nessa camada. Assim, as ligações entre estágios são flexíveis e definidas separadamente da lógica de processamento.
3. Etapa de aglomeração – essa etapa oferece a abstração *Cluster*, que representa um aglomerado, ou um conjunto de estágios da aplicação. O conjunto dos *Clusters* cria uma partição do grafo de estágios. São fornecidos ao desenvolvedor mecanismos para configuração de aglomerados, permitindo que ele indique quais trechos de código precisam ser executados em um ambiente compartilhado e quais dados precisam ser entregues através de interfaces de comunicação remota.
4. Etapa de mapeamento – Essa etapa é responsável por tratar a arquitetura de execução da aplicação com o objetivo de diminuir o impacto da

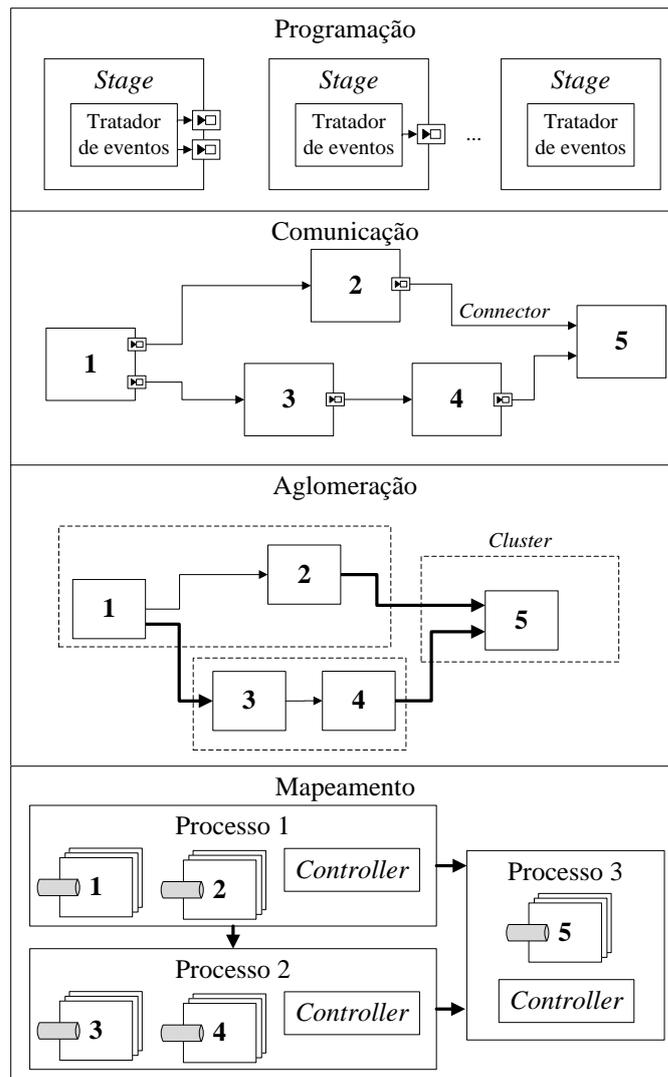


Figura 4.1: Etapas de desenvolvimento Leda.

diversidade de ambientes de execução na qual uma aplicação em estágios pode ser mapeada. A abstração *Controller* controla a política de escalonamento de um processo de sistema operacional mapeado à um único aglomerado da aplicação. Nessa etapa, o usuário pode aplicar otimizações a fim de obter um ambiente de execução que utilize o máximo possível de recursos oferecidos pela máquina que executa cada processo da aplicação.

A seguir, apresentaremos uma visão Geral da API de programação Leda relacionada a cada uma das camadas de desenvolvimento descritas acima. Uma descrição mais detalhada da API de programação Leda é apresentada no Apêndice A.

4.1.2

A API Leda para Programação de Estágios

A biblioteca Leda fornece um pequeno conjunto de primitivas para a criação de estágios. A API Leda inclui um conjunto de funções Lua para especificação de tratadores de eventos e seus atributos. Nesse contexto, eventos representam pacotes de dados, que podem corresponder a uma estrutura de dados específica ou a uma sequência arbitrária de *bytes* que o estágio deve processar.

Um estágio Leda é criado pela primitiva `leda.stage` e é representado por uma tabela Lua que inclui pelo menos um atributo chamado `handler`, associado a uma função Lua que será usada como tratador de eventos. O desenvolvedor implementa um tratador de eventos para cada estágio da aplicação, e este é invocado pelo ambiente de execução Leda sempre que houver novos eventos para processar. A lista de argumentos da função do tratador de eventos define a interface de entrada de eventos do estágio. O tratador de eventos pode, opcionalmente, disparar um ou mais eventos em suas portas de saída.

Instâncias concorrentes de estágios podem ser criadas ou destruídas sob demanda no momento da execução da aplicação. Cada instância paralela do tratador de eventos possui estado interno isolado, tornando a execução do tratador de eventos livre da necessidade de sincronização. Porém, se o tratador realizar chamadas a bibliotecas externas, essas devem prover suas próprias garantias para execução em ambientes com múltiplas *threads*.

A Listagem 4.1 exemplifica a criação de um estágio simples, cujo tratador de eventos apenas imprime os eventos recebidos de sua fila de eventos na saída padrão. Como o desenvolvimento de estágios é desacoplado da estrutura da aplicação, é uma boa prática especificá-los como componentes separados em arquivos independentes.

Listagem 4.1: Exemplo de código de estágio.

```
1 — Arquivo: stage/print.lua
2 require "leda"
3 local stage = leda.stage{
4   — Tratador de eventos
5   handler = function(...) print(...) end,
6   — Define que o estágio é serial
7   serial = true
8 }
9 return stage
```

Na linha 2, importamos a biblioteca Leda para ter acesso às primitivas definidas na API Leda. Em seguida, na linha 3, criamos um estágio.

Em seguida, a linha 5 define um tratador de eventos que imprime quaisquer parâmetros recebidos para a saída padrão do processo onde o estágio irá executar. Atualmente, Leda assume por padrão que estágios podem ser executados com paralelismo interno (com instâncias transientes). No caso do estágio em questão, isso poderia levar a saída de uma instância ser intercalada com a saída de outra instância concorrente de maneira imprevisível, se estiverem executando simultaneamente em *threads* diferentes. Por esse motivo, definimos, na linha 7, o atributo `serial` do estágio com o valor `true` para sinalizar que a execução do tratador de eventos deve ser serializada, e, portanto, no máximo uma instância com estado persistente será criada pelo ambiente de execução para esse estágio. Finalmente, a linha 9 retorna o objeto que representa o estágio recém criado para ser usado durante a definição de um grafo de estágios.

A Listagem 4.2 contém a implementação de um estágio um pouco mais complexo, com duas portas de saída (*line* e *io_error*). O seu tratador de eventos recebe o nome do arquivo para leitura e emite cada linha desse arquivo para a porta de saída *line*.

Listagem 4.2: Código do estágio *lines*.

```

1 — Arquivo: stage/lines.lua
2 require "leda"
3 local stage = leda.stage{
4   handler = function(filename)
5     — Abrir arquivo recebido no evento
6     local file , err = io.open(filename , 'r')
7     if not file then
8       —Enviar erro para a porta 'io_error'
9       assert(leda.send( "io_error " , filename , "Open error: "
10         ..err))
11     return
12   end
13   — Iterar sobre cada linha do arquivo
14   for line in file:lines() do
15     — Enviar linha para a porta de saída 'line'
16     assert(leda.send( "line " , line))
17   end
18 end,
19 — Função de inicialização do estado de instâncias
20 init = function()
21   — Carregar biblioteca 'io' no estado da instância
22   require 'io'
23 end

```

```

23 }
24 return stage

```

A linha 4 da Listagem 4.2 define um tratador de eventos que recebe uma *string* contendo nome o arquivo a ser lido. A linha 6 tenta abrir o arquivo para leitura e, caso a abertura do arquivo falhe, emite um evento contendo o motivo da falha pela porta *io_error* através da primitiva `leda.send` na linha 9. Caso a abertura do arquivo seja bem sucedida, o tratador de eventos do estágio itera sobre cada linha do arquivo e as envia em um evento para a porta de saída *line* na linha 15.

A primitiva `leda.send` é a única interface oferecida para a comunicação com outros estágios de uma aplicação. Seu primeiro parâmetro é uma chave que identifica a porta de saída a ser utilizada para a emissão do evento e os demais parâmetros são empacotados, constituindo o evento a ser colocado na fila de eventos do estágio apropriado pelo ambiente de execução.

Cada estágio pode também definir uma função `bind` contendo um conjunto de verificações a serem realizadas antes da execução da aplicação. Em particular, o código da função `bind` tipicamente faz validações sobre as conexões de portas de saída, uma vez que essas conexões são configuradas externamente. A Listagem 4.3 ilustra essa comodidade apresentando o código de um estágio que filtra *strings* que casam com um padrão definido através de expressões regulares.

Listagem 4.3: Estágio que filtra *strings* que não casam com o padrão definido pela variável `pattern`.

```

1 — Arquivo: stage/match.lua
2 require "leda"
3 local stage = leda.stage{
4   handler = function(str)
5     — Variável pattern é definida no estado interno da
      instância
6     local matches = {string.match(str, self.pattern)}
7     — Se houveram resultados
8     if #matches > 0 then
9       — Enviar resultado da captura da expressão regular
10      assert(leda.send("match", unpack(matches)))
11    end
12  end,
13  init = function()
14    — Carregar a biblioteca 'string' no estado da instância
15    require "string"
16  end,
17  bind = function(self, output)
18    — Garantir a presença da variável pattern

```

```
19     assert(self.pattern, "Pattern field must be defined")
20     — Garantir que a porta 'match' está conectada
21     assert(output.match, "'match' port must be connected")
22     end,
23     pattern='test'
24 }
25 return stage
```

As linhas 4 a 12 definem um tratador de eventos que recebe uma *string* como parâmetro e filtra eventos que não casam com o padrão especificado no atributo interno definido pela variável *self.pattern*. A função `bind`, definida na linha 17, recebe como parâmetros o objeto que representa o estágio criado e uma tabela que contém as portas de saída definidas para o estágio no grafo que irá executar. Ela faz a verificação da presença da porta *match* na tabela de portas de saída recebida, o que indica que ela está ligada a um conector, e a presença do atributo interno *pattern* do estágio.

Por padrão, apenas a biblioteca básica de Lua e a API de Leda são carregadas no ambiente de execução do tratador de eventos de estágios. A função `init` é executada quando uma nova instância é criada, para inicializar seu estado, por exemplo, carregando bibliotecas extras ou definindo funções e variáveis auxiliares em seu estado interno. No caso do estágio definido na Listagem 4.2, essa função é definida na linha 19 e carrega a biblioteca *io* de Lua, necessária para a leitura de arquivos pelo tratador de eventos. No exemplo da Listagem 4.3, a linha 13 define uma função de inicialização que carrega a biblioteca *string*, que contém funções de casamento de padrões.

A primitiva de envio de eventos retorna um valor indicando o sucesso do enfileiramento do evento ou, em caso de falha, o motivo da irregularidade. Por exemplo, filas de eventos podem alcançar uma capacidade máxima, definida pelo escalonador do processo para evitar o consumo ilimitado de recursos de memória. Eventos podem não chegar ao estágio de destino devido a erros de comunicação.

O mecanismo para sinalizar a entrega de eventos pode ser utilizado para definir políticas de controle de taxa de admissão, como implementar prioridades para cada estágio e, em situações de alta demanda, aceitar somente eventos com maior prioridade. Desta forma, estágios podem detectar essas situações e tentar o reenvio de eventos (dependendo-se do erro), descartá-los ou interromper a execução da instância através da função `assert`, resultando em um erro de execução.

4.1.3

Construção do Grafo de Estágios

A API de construção do grafo de estágios permite ao programador configurar os estágios que compõem a nova aplicação e o padrão de comunicação entre eles. Nessa etapa, o desenvolvedor liga as portas de saída para os estágios de destino, criando conectores entre eles. Em tempo de execução, quando um estágio emite um evento para uma porta de saída, o conector correspondente irá direcioná-lo para a fila de eventos apropriada (local ou remota) de acordo com esta configuração.

Dois tipos de conectores são definidos para a composição do grafo: conectores *desacoplados* e conectores *locais* (ou *acoplados*). Estágios conectados através de conectores locais devem necessariamente ser mapeados para um mesmo ambiente de execução, podendo transportar recursos locais. Estágios conectados a conectores desacoplados podem executar em ambientes de execução sem compartilhamento de memória. Essa característica permite ao programador contar com a garantia de que os estágios possam trocar livremente eventos com referências a recursos locais, como descritores de *sockets* ou de arquivos.

A função `leda.connect` cria conectores entre estágios, recebendo quatro parâmetros: O estágio de onde o evento se origina, a porta de saída desse estágio e o estágio consumidor do evento. Caso a porta seja omitida, será usada como porta padrão a porta de índice 1. A função `leda.connect` permite definir o tipo de conector a ser utilizado como quarto parâmetro opcional.

A função `leda.graph` define um grafo de estágios e recebe um conjunto de conectores como parâmetro e retorna um objeto com o grafo formado. Após criado, novos conectores e estágios podem ser adicionados ao grafo através do método `add`.

A Listagem 4.4 contém um código de exemplo que compõe os estágios descritos na seção anterior para criar uma aplicação que imprime para a saída padrão apenas as linhas de arquivos que casam com uma expressão regular. A Figura 4.2 contém uma representação gráfica dessa aplicação.

Listagem 4.4: Exemplo de criação do grafo de estágios de uma aplicação.

```

1 — Arquivo: app.lua
2 require "leda"
3 local lines = require "stage.lines" — Listagem 4.2
4 local matcher = require "stage.match" — Listagem 4.3
5 local printer = require "stage.print" — Listagem 4.1
6
7 — Cria o grafo de estágios da aplicação e seus conectores
8 local app = leda.graph{
```

```

9   leda.connect(lines , "line " ,matcher , "local " ) ,
10  leda.connect(matcher , "match " ,printer )
11 }
12 — Retorna o grafo da aplicação
13 return app

```



Figura 4.2: Representação gráfica da aplicação definida na Listagem 4.4.

Nesse exemplo, as linhas 3 a 5 importam estágios que serão usados na aplicação. A linha 9 define um conector do estágio *lines* para o estágio *matcher* através da porta *line*, cujo tipo é *local*. Já a linha 10 define um conector desacoplado (tipo padrão de conector) entre a porta *match* do estágio *matcher* para o estágio *printer*.

4.1.4 Aglomeração, Mapeamento e Execução

O objetivo da etapa de aglomeração é equilibrar a carga atribuída a cada unidade de execução disponível, procurando balancear a granularidade de estágios e o custo de comunicação dentro de aglomerados. O processo de aglomeração divide o grafo da aplicação para agrupar estágios em unidades de execução. O conjunto de aglomerados deve formar uma partição do grafo, ou seja, deve contemplar todos os estágios definidos na aplicação e cada estágio deve estar associado a apenas um aglomerado.

O agrupamento de estágios deve obedecer os padrões de comunicação definidos pelos conectores do grafo de estágios, ou seja, os estágios que se comunicam através de conectores locais devem residir no mesmo aglomerado. Aglomerados são mapeados a unidades de execução exclusivas. Estágios em aglomerados diferentes se comunicam através de mecanismos de troca de mensagens para filas de eventos remotas.

O código da Listagem 4.5 contém um exemplo de partição do grafo de estágios da aplicação descrita na seção anterior.

Listagem 4.5: Exemplo de aglomeração e mapeamento.

```

1 require 'leda '
2 — importa a definição do grafo da aplicação
3 local graph = require 'app' — Listagem 4.4
4 — importa a definição dos estágios
5 local printer = require 'stage.print' — Listagem 4.1
6 local matcher = require 'stage.match' — Listagem 4.3
7 local lines = require 'stage.lines' — Listagem 4.2

```

```
8 — Cria primeiro aglomerado contendo apenas o estágio printer
9 local cluster1 = leda.cluster(printer)
10 — Cria segundo aglomerado contendo o resto da aplicação
11 local cluster2 = graph:all() - cluster1
12 — Define o particionamento do grafo
13 graph:part(cluster1, cluster2)
14 — Mapeamento dos aglomerados a processos
15 graph:map("host1:porta1", "host2:porta2")
16 — Cria atributo pattern no estágio matcher
17 matcher.pattern = arg[1]
18 for i=2,#arg do
19     — Envia evento para o estágio lines
20     lines:push(arg[i])
21 end
22 — Execução do grafo
23 graph:run()
```

A partição do grafo da aplicação em aglomerados é realizada através da invocação do método `part` do grafo criado previamente. Esse método recebe os aglomerados criados pela primitiva `leda.cluster` da API de Leda. O método utilitário `all`, usado na linha 11 da Listagem 4.5, retorna um aglomerado contendo todos os estágios definidos no grafo. Aglomerados são representados como conjuntos de estágios e podem ser manipulados através dos seguintes operadores: união (+), diferença (-) ou interseção (*). No exemplo da Listagem 4.5, na linha 13, a aplicação é particionada em dois aglomerados: um contendo um único estágio (*printer*) e o outro com o resto dos estágios da aplicação.

Antes da execução da aplicação propriamente dita, pode ser necessário definir algum atributo interno de estágios, como por exemplo, o atributo *pattern* do estágio *matcher*, que é verificado pela sua função `bind`. Isso é realizado na linha 17 da Listagem 4.5, que assume que o primeiro argumento passado ao programa é uma *string* contendo uma expressão regular para ser usada pelo tratador de eventos do estágio. Ainda antes da execução, o pipeline recebe eventos para iniciar os fluxos de processamento concorrentes dentro da aplicação. O *loop* da linha 18 enfileira o restante dos parâmetros passados ao programa na fila de eventos do estágio *lines* (método *push*), e constituem os seus eventos iniciais para processamento.

Em seguida, os aglomerados são mapeados para unidades de execução através do método `map` que associa cada aglomerado a um processo de sistema operacional identificado pelo seu endereço de rede e a porta na qual aguarda por conexões. Processos de sistema operacional devem ser criados a priori através da primitiva `leda.start`, como apresentado no código da Listagem 4.6.

Listagem 4.6: Instanciando processos Leda ajudantes.

```
1 require 'leda '  
2 local c = require 'leda.controller.thread_pool'.get(10)  
3 leda.start{controller = c, host = 'host1', port = 9999}
```

A primitiva `leda.start` recebe como parâmetro o controlador a ser usado no processo. No exemplo, é usado um controlador contendo um *pool* de *threads* fixo com 10 *threads* de sistema operacional. Atualmente, caso o controlador não seja especificado, a implementação utiliza por padrão um controlador que instancia um *pool* de *threads* fixo com uma *thread* por CPU presente na máquina.

Finalmente, a linha 23 da Listagem 4.5 inicia a execução propriamente dita da aplicação. O método `run` pode receber o mesmo conjunto de parâmetros usados pela primitiva `leda.start`.

4.2

Comunicação e Gerenciamento de Dados

A arquitetura de execução de Leda dá suporte a dois tipos de comunicação entre estágios: comunicação intra-processo (entre estágios que residem em um mesmo aglomerado, tratados pelo mesmo processo) e inter-processo (entre estágios mapeados a processos distintos). Os dois tipos de comunicação se comportam de maneira uniforme, ou seja, eventos são entregues às filas de eventos correspondentes em ambos os casos. Porém, estágios em um mesmo aglomerado podem se beneficiar do acesso a recursos compartilhados para otimizações de comunicação e particionamento de trabalho sobre estruturas internas como arquivos ou *sockets*.

A comunicação entre instâncias é feita copiando-se os dados a serem enviados a partir da pilha ou *heap* do remetente para a fila de eventos do receptor. A cópia pode ser enfileirada diretamente em memória se o estágio de destino estiver no mesmo aglomerado, ou através de conexões TCP, se o destino for remoto. Os dados do evento devem ser desenfileirados e carregados na pilha de destino antes da execução da instância. O coletor de lixo do remetente é livre para desalocar a memória associada aos dados imediatamente após o envio, uma vez que eles foram copiados. Porém, a necessidade de cópia dos dados torna o custo de comunicação proporcional ao tamanho da mensagem. Esse tipo de operação de comunicação é proibitivo em estágios que lidam com uma grande quantidade de dados, como em *pipelines* de processamento de imagens ou de vídeos.

Há uma relação de compromisso entre o isolamento de estágios e a comunicação, pois quanto mais isolados são os ambientes de execução desses

estágios, potencialmente mais alto é o custo da comunicação (Fahndrich2006), por exemplo, pela necessidade de serialização e a sobrecarga de empacotamento dos dados. Instâncias de estágios são isoladas e, portanto, alocam e gerenciam sua área de memória privada sem acesso a um estado global compartilhado. Cada instância possui pilha de procedimento e *heap* próprias. A pilha local é usada para armazenar parâmetros de funções e variáveis locais, enquanto seu estado local é armazenado na *heap*.

Para impedir a degradação de desempenho de estágios nesses cenários, a implementação de Leda usa a noção de referência única a dados (Haller2010), que consiste em mover os dados a partir da região de memória do emissor para a região de memória do receptor. Assim, dados transmitidos não estarão mais acessíveis para o remetente após terem sido enviados, logo não há condições de corrida. Nesses casos, somente o coletor de lixo da instância de destino é responsável por desalocar e finalizar o objeto relacionado ao dado. No entanto, essa técnica só pode ser aplicada em instâncias de estágios que se comunicam através de conectores locais, e, portanto, devem executar em um mesmo aglomerado. Essa técnica também permite que referências a recursos locais possam ser trocadas através de conectores locais. A comunicação através de conectores desacoplados requer que dados sejam sempre serializados (ou copiados) para transmissão.

Conectores desacoplados serializam automaticamente valores dos seguintes tipos primitivos de Lua: `nil`, *string*, números, booleanos, tabelas e funções. Valores de tipos de usuário (*userdata*) devem definir uma função `_persist` para implementar a serialização e reconstrução desses valores, e possibilitar o transporte em conectores desacoplados. Similarmente, o programador pode definir uma função `_wrap` para implementar o mecanismo de passagem de referências para esses valores em conectores locais. Nesse caso, a semântica de referência única deve ser garantida pela implementação da função para evitar o compartilhamento de dados entre instâncias distintas. Leda fornece funções predefinidas para passagem de referências únicas a descritores de arquivo, *sockets* e ponteiros para *buffers* de memória.

4.3

Implementação de Leda

Nessa seção, descrevemos a implementação do ambiente de execução de instâncias de estágios e dos componentes internos da biblioteca Leda.

A próxima subseção descreve a implementação do ambiente de execução de instâncias de estágios e a subseção 4.3.2 descreve a estrutura de execução de processos Leda.

4.3.1

Gerenciamento de Instâncias

Instâncias são criadas sob demanda pelo ambiente de execução para processamento dos eventos da fila do estágio até um limite máximo definido pelo processo no qual o estágio executa. Durante a criação de uma instância, o código do tratador de eventos é carregado e a função `init` do estágio é executada para inicializar seu estado interno.

O modelo proposto requer que instâncias de estágios possuam estados isolados, portanto, um mecanismo para isolamento de memória deve ser provido pela implementação. A implementação padrão de Lua interpreta *scripts* em uma máquina virtual (VM) e permite que várias instâncias de sua VM executem independentemente dentro de um processo de sistema operacional. Cada instância da VM é comumente chamada de *estado Lua* (Ierusalimschy2006). Os estados Lua não compartilham informações, ou seja, cada um possui o seu próprio conjunto de funções, variáveis globais e estruturas de controle.

A implementação garante o isolamento de instâncias através do uso de estados Lua exclusivos. A API C de Lua nos permite divorciar estados Lua de *threads* de sistema operacional: o desenvolvedor pode executar cada estado Lua dentro de uma *thread* de sistema operacional diferente ou compartilhar um número limitado de *threads* entre um número arbitrário de estados Lua (Skyrme2008).

Como mencionado anteriormente, Lua oferece concorrência cooperativa por meio de co-rotinas. Cada co-rotina possui sua própria pilha de execução e compartilha o estado global com as demais co-rotinas dentro de um estado Lua, porém a troca de contexto é realizada de forma explícita.

Instâncias exploram o mecanismo de co-rotinas para compartilhar as *threads* do *pool* e prover concorrência em tratadores de eventos. Instâncias podem, então, ser desanexadas de *threads* durante a execução e continuadas em um momento posterior. Esse mecanismo é utilizado para prover interfaces de entrada e saída assíncronas ou usar temporizadores em tratadores de eventos.

Dada a independência dos estados Lua, instâncias diferentes possuem suas próprias variáveis globais e não são capazes de compartilhar variáveis, o que garante a ausência de condições de corrida, uma vez que apenas uma *thread* está associada a uma instância durante sua execução.

Apesar da criação de um estado Lua ser uma operação barata, carregar todas as bibliotecas de Lua pode levar mais de dez vezes o tempo necessário para criar um estado (Ierusalimschy2006). Assim, para reduzir o custo do processo de criação de estados Lua, apenas a biblioteca padrão de Lua e uma API mínima para instâncias são automaticamente carregadas para cada nova

instância criada. As bibliotecas restantes de Lua (*io*, *os*, *table*, *string*, *math*, e *debug*) são pré-registradas e podem ser carregadas normalmente em instâncias através da função `require` de Lua.

A Figura 4.3 contém um diagrama de transição com os possíveis estados de instâncias durante seu ciclo de vida.

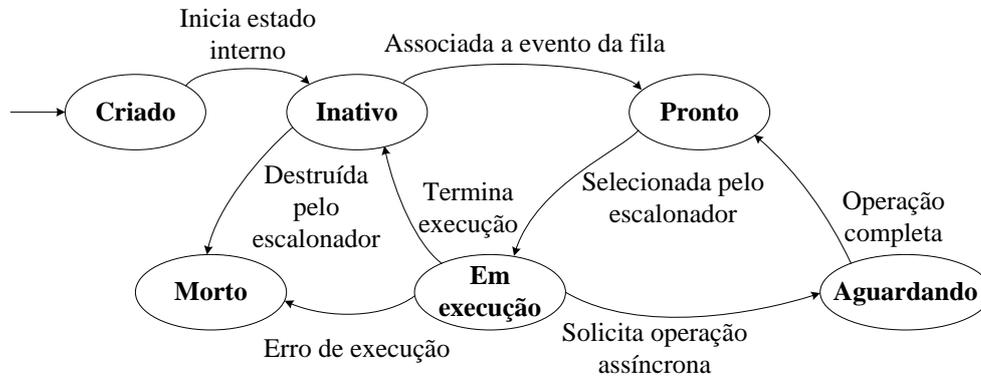


Figura 4.3: Diagrama de transição de estados de instâncias.

Em um dado momento, uma instância de estágio pode estar em um de seis possíveis estados de execução. A instância inicia no estado *Criado*, onde seu estado Lua interno está vazio e, portanto, pode se especializar em uma instância de qualquer estágio. Após a função de inicialização de um estágio ser executada em seu estado interno, a instância passa ao estado *Inativo*, significando que ela está esperando por eventos da fila de seu estágio. Após ser vinculada a um evento da fila, ela passa ao estado *Pronto*, e aguarda em uma fila de prontos. Ao ser selecionada para execução pelo escalonador do processo, a instância passa ao estado *Em execução*, onde permanece associada a uma *thread* de sistema operacional até finalizar seu processamento, voltando ao estado *Inativo*. Como os tratadores de eventos executam em co-rotinas, é possível desanexá-los de *threads* conforme necessário. Nesse caso, a instância vai para o estado *Aguardando* e libera cooperativamente a *thread* de volta ao *pool* do processo. Após a operação assíncrona solicitada ser atendida, a instância volta ao estado *Pronto* para continuar o processamento.

Para garantir a ausência de condições de corrida no caso dos estágios com estado persistente, apenas uma instância é criada para eles, portanto, eventos emitidos para estes estágios são processados em série.

Tratadores de eventos executam em um ambiente de execução protegido, onde o escalonador captura falhas e erros de execução em tratadores de eventos. Em caso de erros, a instância em execução entra no estado *Morto*. Nesse caso, a sua pilha de execução é analisada e impressa na saída de erro do processo e, então, é destruída, resultando na finalização e liberação de quaisquer recursos referenciados em seu estado interno pelo coletor de lixo de Lua. Além disso,

uma instância inativa pode ser destruída pelo escalonador do processo, por exemplo, com o propósito de liberar recursos de memória para outras instâncias conforme a política adotada para alocação de recursos.

4.3.2

Processo Leda

A unidade básica de execução de aglomerados de estágios definidos em uma aplicação é um processo de sistema operacional com múltiplas *threads* que executa a biblioteca Leda (ou *processo Leda*). Os diferentes processos que compõem o ambiente de execução de Leda podem executar em máquinas arbitrárias ligadas por uma infraestrutura de rede ou em uma mesma máquina local.

Cada processo Leda possui um identificador único e a comunicação entre esses processos é feita via canais TCP. O identificador do processo contém o endereço IP e porta TCP na qual ele aguarda por eventos externos. Os eventos Leda são pacotes de dados binários contendo valores Lua. O ambiente de execução do processo Leda é responsável por converter valores Lua em sequências de *bytes* para transmissão a outros processos de forma transparente para a aplicação, caso necessário. Porém, como discutido na seção 4.2, a comunicação em memória compartilhada é permitida para estágios que executam em um mesmo processo Leda.

A implementação de Leda recorre a uma arquitetura de componentes que formam o ambiente de execução do sistema. A Figura 4.4 ilustra os componentes internos de nossa implementação.

O componente *Aplicação* contém o resultado dos *scripts* de configuração definidos pelo desenvolvedor nas etapas de programação da aplicação. Isso inclui informações sobre a descrição declarativa do grafo de estágios, o código de tratadores de eventos, funções internas e variáveis auxiliares de estágios em formato binário de *bytecode* Lua, assim como a especificação dos aglomerados definidos para a aplicação e os identificadores dos processos no qual os aglomerados estão mapeados.

Cada processo da aplicação começa a sua execução lendo a descrição declarativa de seu respectivo aglomerado e carrega o código dos estágios correspondentes no componente *Aglomerado*. Quando necessário, a especificação imperativa dos estágios pode ser instanciada pelo componente *instâncias* do processo. Nesse componente, técnicas de compilação *just in time* podem ser usadas para se converter o código binário Lua para código nativo e obter ganhos de desempenho através do uso de instruções específicas para a arquitetura de execução em questão.

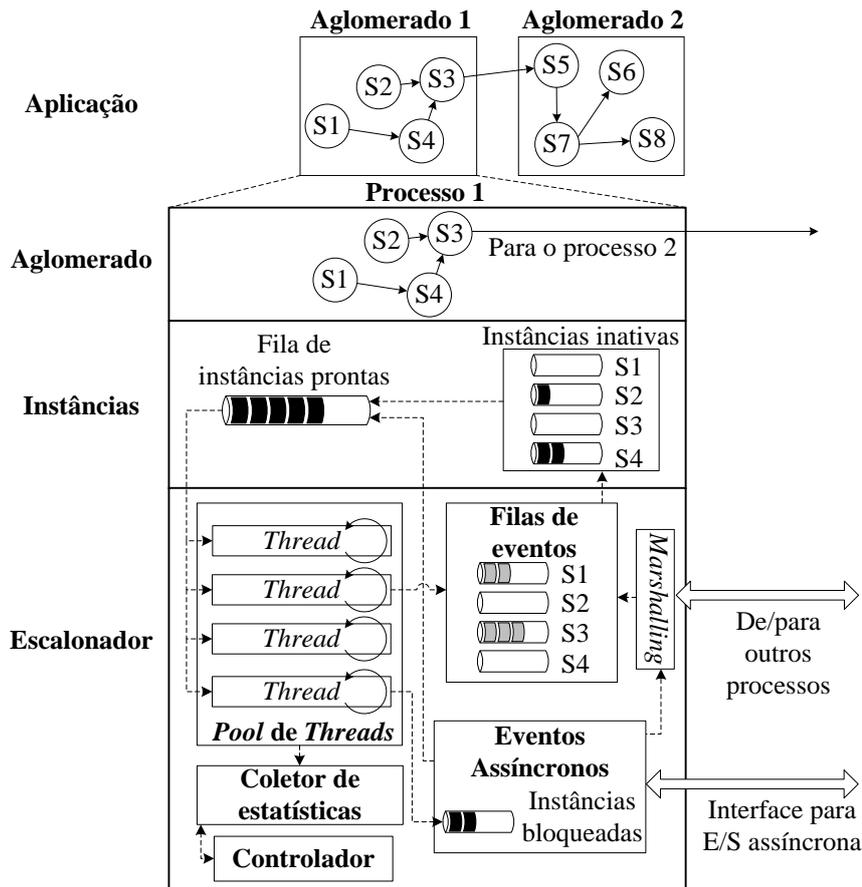


Figura 4.4: Componentes internos de um processo Leda.

O componente *escalonador* gerencia o ciclo de vida de instâncias de acordo com o diagrama de estados da Figura 4.3. Instâncias inativas de estágios são associadas pelo escalonador a eventos pendentes específicos de sua fila eventos e são postas na fila de instâncias prontas para executar no processo. *Threads* do *pool* constantemente retiram uma instância da fila de prontas e a executa atomicamente até o seu término, se deparar com um erro de execução ou ceder cooperativamente o controle para efetuar operações assíncronas.

Filas em memória compartilhada guardadas por travas de exclusão mútua podem adicionar uma sobrecarga de sincronização considerável quando sobrecarregadas devido a contenção de recursos de processamento (Silvestre2009). Decidimos então usar estruturas livres de bloqueios para a nossa implementação das filas dentro de processos. Utilizamos a biblioteca *Threading Building Blocks* (TBB) em sua versão mais atual (4.1 Update 3) (Reinders2010), desenvolvida pela Intel, que disponibiliza um conjunto de estruturas de dados para concorrência que utiliza instruções *Compare and Swap* (CAS). Mais especificamente, usamos as estruturas `concurrent_bounded_queue` para as filas de eventos, e a estrutura `atomic` para contadores compartilhados.

Para atingir a meta de suportar alta concorrência com gerenciamento

de tarefas cooperativa com múltiplas *threads*, interfaces eficientes para entrada e saída assíncrona são fornecidas pelo módulo de *Eventos Assíncronos* do componente *escalonador*. Esse módulo possui uma *thread* de sistema operacional exclusiva para fornecer uma implementação de entrada e saída assíncrona para *sockets*, temporizadores e sinais, que são abstraídos pela biblioteca *libevent* (Russell2012), fornecendo suporte a interfaces de eventos de alto desempenho nos principais sistemas operacionais e aumentam a portabilidade do sistema de eventos assíncronos. Porém, a implementação atual para entrada e saída assíncrona para o sistema de arquivos usa a interface *AIO* (Robbins2003), cujo suporte é exclusivo para sistemas operacionais que usam o *kernel* Linux, permitindo que leituras e escritas em disco aconteçam em segundo plano. O módulo de *Eventos Assíncronos* também é responsável por gerenciar a comunicação inter-processo com o auxílio do módulo *Marshalling*, cuja função é prover uma interface unificada para a codificação valores binários entre processos.

Um aspecto importante da arquitetura em estágios é o desacoplamento entre a especificação da lógica de processamento da aplicação e a política de gerenciamento de recursos. Assim como em SEDA, nossa implementação provê essa funcionalidade através de controladores dinâmicos de recursos, que procuram manter o sistema dentro de limites de operação, apesar das flutuações da carga percebida. A implementação do processo Leda inclui um módulo *coletor de estatísticas* que registra informações sobre uso de memória, comprimentos das filas, latências de processamento e de comunicação entre estágios durante a execução. Os dados gerados pelo *coletor de estatísticas* podem ser utilizados para visualizar o comportamento dinâmico e o desempenho da aplicação. Por exemplo, um gráfico do comprimento das filas ao longo do tempo de execução da aplicação pode ajudar a identificação de gargalos na *pipeline*.

O *coletor de estatísticas* coleta periodicamente as estatísticas de execução de estágios e as disponibiliza ao controlador. Essas informações podem ser escritas em um arquivo de *log* e visualizadas posteriormente usando ferramentas comuns para esse fim, como *gnuplot* e *excel*. Mais especificamente, o *coletor de estatísticas* disponibiliza informações sobre uso atual de memória, tamanho e limite máximo das filas de eventos e da fila de instâncias prontas, tamanho do *pool* de *threads*, número de *threads* ativas e o número de instâncias de estágios que estão em execução, bem como a latência média de execução de instâncias e a latência de comunicação em conectores, número total de eventos emitidos por estágios e a vazão média de emissão de eventos por conectores em eventos por segundo. Com base nessas informações, o controlador pode tomar decisões de escalonamento que refletem o estado atual da carga de processamento do

processo.

5 Avaliação

Este capítulo descreve a implementação e análise de seis aplicações que exercitam as características do modelo proposto no Capítulo 3 e refletem diferentes propriedades relevantes no contexto deste trabalho. Além do modelo, avaliaremos alguns aspectos específicos da implementação de Leda, apresentada no Capítulo 4.

Classificaremos as avaliações apresentadas neste capítulo em três categorias principais. Primeiramente, analisaremos a flexibilidade proporcionada pelo modelo de programação, avaliando a aplicabilidade do modelo no desenvolvimento de programas paralelos e o impacto de diferentes configurações de uma mesma aplicação em ambientes de execução distintos. Em seguida, apresentaremos a avaliação da implementação do modelo, comparando a sobrecarga de processamento e comunicação de aplicações desenvolvidas em Leda com outras soluções semelhantes. Finalmente, avaliaremos subjetivamente a facilidade de uso de Leda e os mecanismos de reuso de estágios através da descrição da implementação de duas aplicações não convencionais.

Nossa avaliação experimental foi realizada em um *cluster*¹ com memória distribuída composto por 32 máquinas contendo processadores Intel Xeon X5650 2,66GHz com 24 núcleos e 24GB de RAM. As máquinas são interligadas por uma rede InfiniBand com banda máxima de 10Gbits/s e executam o sistema operacional Linux, com *kernel* versão 2.6.32 64 bits. O compilador usado foi o GCC versão 4.3.4.

5.1 Flexibilidade do Modelo

Para avaliar a flexibilidade do modelo, duas aplicações foram consideradas: uma aplicação para processamento de imagens, que utiliza processamento em *pipeline* e apresenta grande carga de processamento e comunicação, e uma aplicação de processamento complexo de eventos, para processamento e correlação de fluxos de eventos de fontes distribuídas.

¹O *cluster* usado foi cedido pelo Instituto Tecgraf de Desenvolvimento de Software Técnico-Científico da Pontifícia Universidade Católica do Rio de Janeiro.

A primeira aplicação exercita os mecanismos de comunicação e compartilhamento de dados do modelo proposto, além de explorar a facilidade de configuração de diferentes ambientes de execução, enquanto a segunda aplicação exercita a flexibilidade do modelo com a utilização de diferentes configurações em um mesmo ambiente de execução.

5.1.1

Pipeline de Processamento de Imagens

Nessa aplicação de teste, implementamos um *pipeline* para processamento de imagens. Essa aplicação apresenta algumas características que são adequadas para a estrutura de processamento em estágios. O processamento de imagens normalmente não necessita de estado compartilhado, pois cada operação atua exclusivamente sobre o conjunto de dados de entrada de forma independente. As operações envolvidas são computacionalmente complexas e de fácil paralelização, beneficiando-se de cenários de execução com múltiplos processadores.

O objetivo dessa aplicação é tratar um número de imagens armazenadas em disco, distribuindo os recursos de processamento disponíveis para maximizar a vazão de eventos em cada estágio, onde a vazão é definida pelo número de eventos emitidos por um estágio em um determinado período de tempo. Eventos carregam uma representação da imagem que deve ser tratada em cada estágio.

O grafo da aplicação de processamento de imagens consiste de uma série de estágios que implementam os diferentes filtros para imagens e estágios específicos para compressão e expansão do formato de representação da imagem. A Figura 5.1 apresenta o grafo de estágios da aplicação e a divisão de aglomerados realizada em três cenários de execução.

Os estágios *image decode* e *image encode* realizam, respectivamente, a expansão e compressão do formato de representação das imagens através da biblioteca *Imlib* (*Imlib2004*), que também efetua as operações de leitura e escrita dos arquivos no disco. Nos testes foram utilizadas imagens comprimidas no formato JPEG. Cada imagem passa por três estágios que aplicam filtros distintos, implementados em C, antes de ser comprimida novamente para o formato original. O estágio *denoise* aplica o filtro de redução de ruídos através do algoritmo de mediana seletiva. O estágio *color correction* ajusta as tonalidades de cores e de luminância através da expansão do histograma em cada componente de cor da imagem. Finalmente, o estágio *scale* gera uma cópia de menor resolução da imagem, que é armazenada separadamente da imagem original.

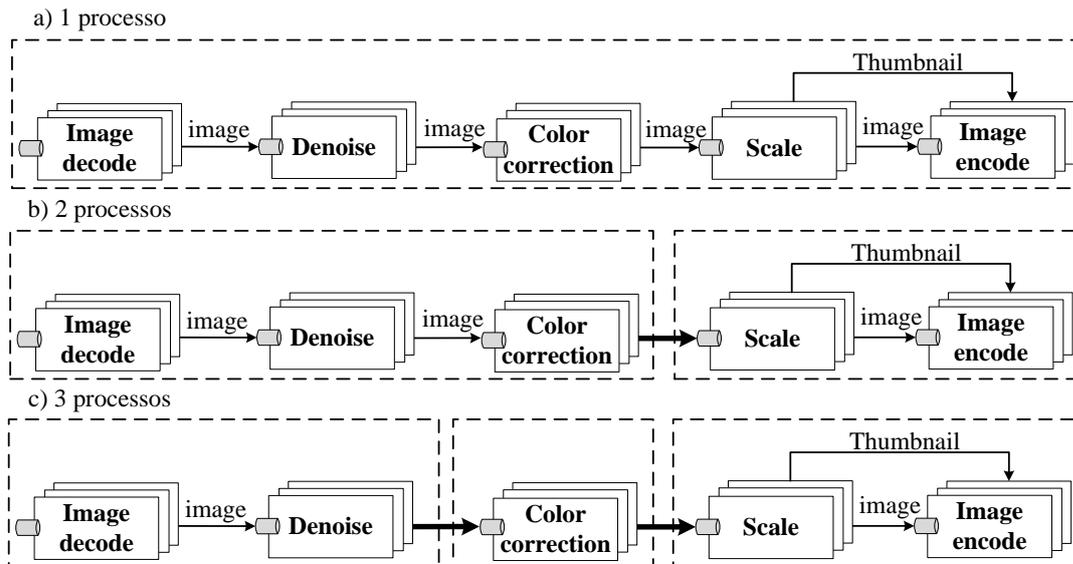


Figura 5.1: Grafo de estágios da aplicação de processamento de imagens e a divisão de aglomerados em três cenários de execução.

Tabela 5.1: Tamanho das imagens e a tamanho total das 500 imagens usadas no teste.

<i>Caso de teste</i>	<i>Tamanho das imagens</i>	<i>Total da carga de trabalho</i>
Baixa resolução	260 kilopixels	33,7 MB
Média resolução	1 megapixel	100,15 MB
Alta resolução	8 megapixels	1550 MB

Os testes foram realizados em três cenários de execução com diferentes configurações de aglomerados ilustrados na Figura 5.1. Em (a), todos os estágios da aplicação foram aglomerados em um único processo de sistema operacional. Em (b), o grafo da aplicação foi particionado em dois aglomerados que foram mapeados para processos localizados em máquinas distintas do ambiente de execução. Em (c), o grafo foi particionado em três aglomerados, cada qual mapeado para um processo localizado em uma máquina distinta.

A carga de processamento dos filtros é proporcional à resolução espacial (quantidade de *pixels*) das imagens. Portanto, para avaliar diferentes granularidades de comunicação e processamento, nós optamos por usar três tipos de casos de teste, utilizando imagens de baixa, média e alta resolução. Os testes foram realizados através da medição do tempo total de processamento de uma carga de trabalho composta por 500 imagens em cada caso. A Tabela 5.1 mostra as características das imagens utilizadas para cada caso de teste.

Os filtros de processamento de imagens são computacionalmente intensivos, e são limitados principalmente pela capacidade de processamento das máquinas usadas, assim, a execução da aplicação foi realizada utilizando uma *thread* por processador disponível no ambiente de execução. O cenário de teste

com 1 processo utilizou 24 *threads*, o cenário com 2 processos utilizou no total 48 *threads* e o cenário com 3 processos utilizou o total de 72 *threads*.

A Figura 5.2 contém os gráficos com os resultados dos testes de acordo com os tempos totais de processamento da aplicação em cada caso de teste para os três cenários de execução.

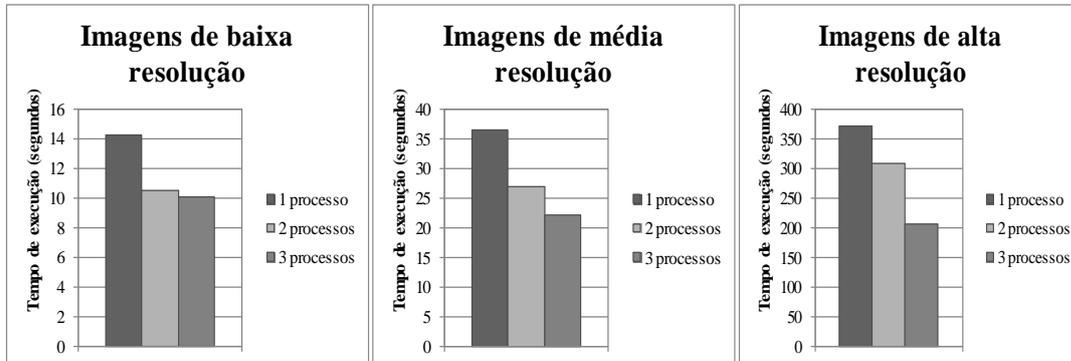


Figura 5.2: Tempos totais de execução da aplicação de processamento de imagens nos casos de teste com imagens de baixa, média e alta resolução de acordo com os cenários de execução com 1, 2 e 3 processos.

Analisando os resultados dos cenários de teste, é possível verificar que os casos de imagens de alta resolução apresentam maiores ganhos de desempenho com a distribuição dos estágios em ambientes distintos. O tempo total de processamento do cenário com 3 processos foi aproximadamente 30% menor do que o do cenário com 1 processo no caso de baixa resolução e 46% menor no caso de alta resolução. Esse ganho pode ser explicado pelo mecanismo de comunicação assíncrono aliado à carga de processamento dos filtros utilizados. Apesar de imagens de alta resolução apresentarem um custo de comunicação superior, a carga de processamento dos filtros também aumenta de acordo com a quantidade de *pixels* da imagem. Como os dados são transmitidos através da rede em paralelo com a execução da aplicação, o tempo perdido na comunicação é compensado pelo tempo de execução de estágios.

Por outro lado, os filtros aplicados a imagens com baixa resolução não apresentam uma carga de processamento suficiente para compensar o tempo de comunicação nos cenários em que temos processos distribuídos. Assim, os ganhos de desempenho para imagens de baixa resolução nos casos de 2 e 3 processos são menos expressivos com relação aos ganhos desses cenários nos casos de teste com média e alta resolução.

Para ajudar a explicar os ganhos de desempenho nos cenários com processos distribuídos é interessante analisar o estado interno de execução dos estágios da aplicação. A Figura 5.3 apresenta os gráficos contendo as

estatísticas de execução extraídas do módulo “coletor de estatísticas” de Leda para os estágios da aplicação no caso de teste com imagens de média resolução.

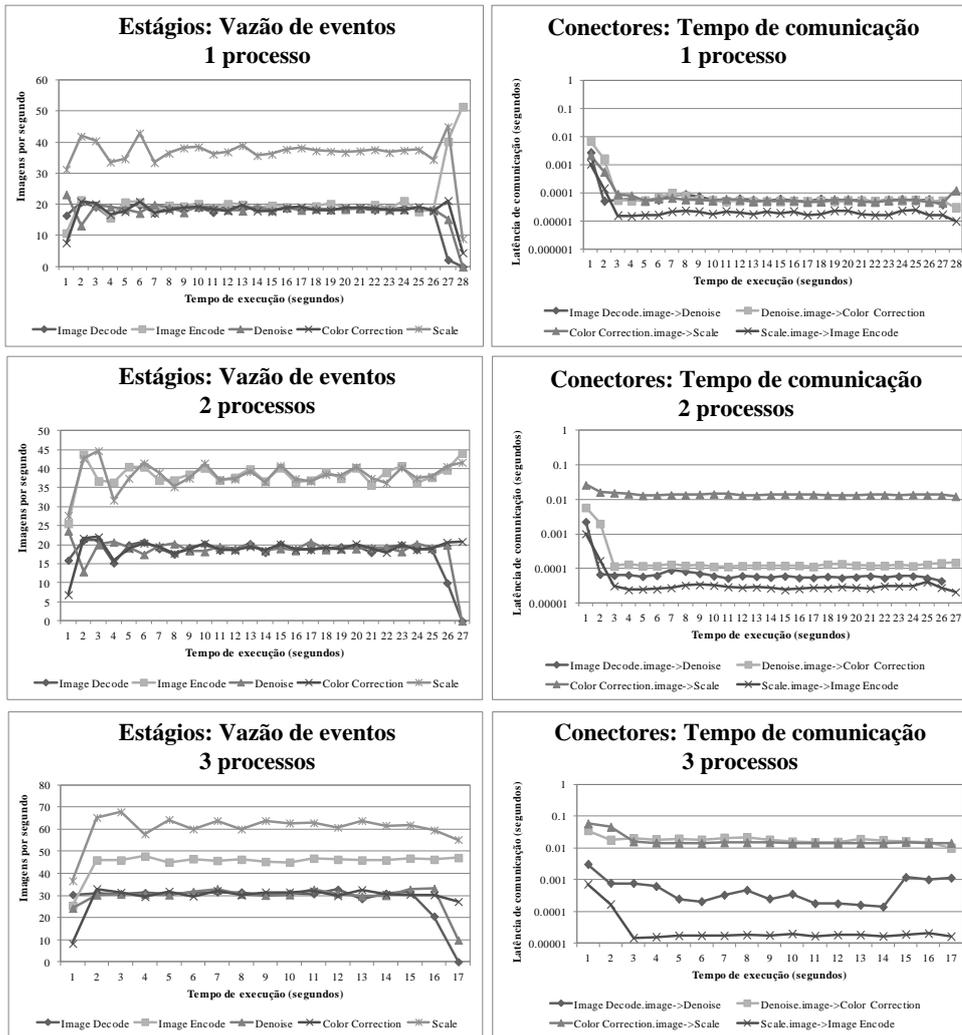


Figura 5.3: Estatísticas de execução da aplicação de processamento de imagens para o caso de teste com imagens de média resolução.

A análise das estatísticas de execução dos estágios nos permite observar que houve um aumento de aproximadamente 50% na vazão de eventos dos estágios de maior processamento entre o cenário de 1 processo para o cenário com 3 processos.

É interessante observar que o custo comunicação em conectores que ligam estágios mapeados em processos distintos é aproximadamente 100 vezes maior do que em conectores que ligam estágios mapeados no mesmo processo. Esse comportamento deve-se ao fato de que estágios em um mesmo aglomerado fazem uso da passagem de referências únicas, enquanto estágios em processos distintos precisam se comunicar através da passagem de imagens sem compressão pela rede. Porém, o custo de comunicação foi compensado pela carga de processamento dos filtros de imagens aplicados nos estágios da aplicação.

A flexibilidade para configurar o ambiente de execução da aplicação aliado aos dados fornecidos pelo módulo de estatísticas dão o suporte para encontrar a configuração que resulta em um desempenho satisfatório de acordo a plataforma de execução em questão.

5.1.2

Processamento Complexo de Fluxos de Eventos

Sistemas para processamento complexo de eventos (Wu2006) constituem uma classe emergente de aplicações que processam dados de diversas fontes, muitas vezes distribuídas, e que geram um grande volume de eventos ou que extraem eventos de grandes janelas. Por exemplo, aplicações de monitoramento de ofertas de ações na bolsa de valores muitas vezes aplicam uma janela deslizando (por exemplo, nas últimas 12 horas) para uma sequência de eventos de interesse. Em muitos cenários, tais janelas são grandes e os eventos relevantes para a consulta são muito dispersos com os outros através da janela. Ao contrário da detecção de eventos simples, que relata apenas a ocorrência de um evento em particular, a extração de eventos relevantes para os resultados possíveis provoca aumento significativo na carga de processamento.

Consultas de eventos complexos podem abordar tanto as ocorrências e não-ocorrências de eventos, e impor restrições temporais (por exemplo, a ordem de ocorrências em janelas deslizando), bem como restrições baseadas em valores sobre esses eventos. Esses sistemas podem exigir que tais eventos sejam filtrados e correlacionados para detecção de padrões complexos e transformados em novos eventos que alcançam um nível semântico adequado para aplicações finais.

Este teste procura explorar a flexibilidade de nosso modelo em um experimento onde o objetivo é encontrar a melhor configuração para o ambiente de execução de uma aplicação de processamento de fluxos de eventos a partir de fontes distribuídas. A Figura 5.4 contém a estrutura de uma aplicação sintética composta por sete tipos de estágios.

O primeiro cenário de execução, representado pelos retângulos tracejados da Figura 5.4, possui três fontes de eventos (estágio *fonte*) extraídas de *logs* de acesso do servidor *web* Apache. Cada fonte é associada a um aglomerado exclusivo que é mapeado a um processo que executa junto ao servidor *web*. O estágio *janela* agrupa eventos recebidos em uma janela espacial de tamanho pré-definido e a envia como um vetor. O estágio *copiar* efetua cópias do vetor de eventos recebidos para serem processadas pelos estágios *persiste*, que faz a escrita do vetor em disco, *correlaciona*, que faz a co-relação entre cada evento da janela, e *filtra*, que identifica eventos relevantes da janela, descartando os

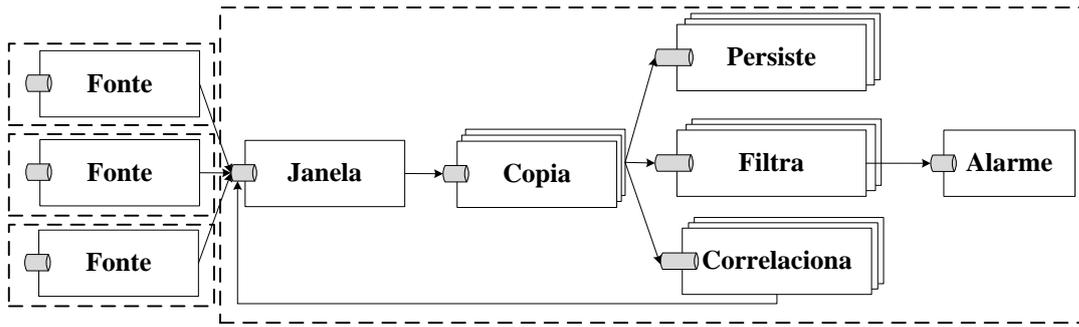


Figura 5.4: Particionamento do cenário 1 para o grafo de estágios da aplicação de processamento complexo de eventos.

demais. Caso um evento relevante esteja presente na janela, este é repassado para o estágio *alarme*, que executa uma ação sobre o evento recebido.

Essa aplicação pode ser configurada de diversas formas para execução. Todos os estágios de processamento de eventos podem ser aglomerados em um processo de sistema operacional, compartilhando um único *pool* de *threads*, como no caso do cenário 1 da Figura 5.4. Outra possibilidade é dividir esses estágios em mais de um aglomerado, separando-se o *pool* de *threads* em partes específicas da aplicação. Além da configuração de aglomerados, Leda pode configurar um número variável de instâncias para cada estágio, permitindo uma equalização da divisão de *threads* entre os estágios em um mesmo aglomerado.

O estágio *correlaciona* é o estágio que possui maior carga de processamento, pois executa uma operação com complexidade quadrática com relação ao tamanho do vetor recebido, enquanto os estágios *persiste* e *filtra* possuem complexidade linear. Portanto, definimos um segundo cenário, onde o estágio *correlaciona* é isolado dos demais. Esse cenário é ilustrado na Figura 5.5.

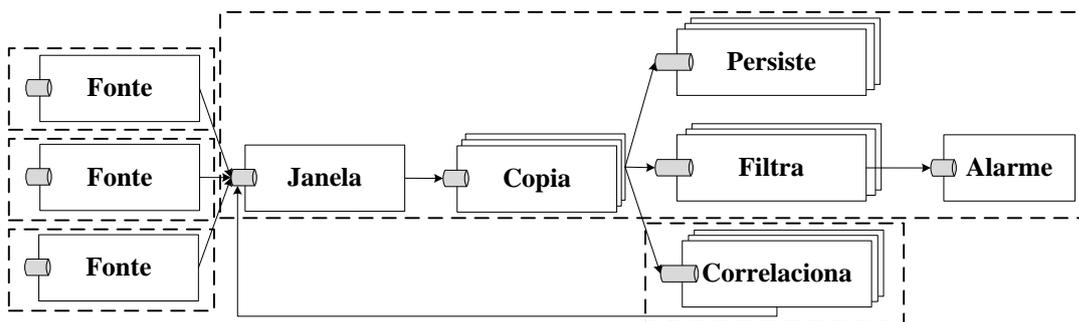


Figura 5.5: Particionamento do cenário 2 para o grafo de estágios da aplicação de processamento complexo de eventos.

Os testes foram realizados através do tratamento de arquivos de *log* pré-armazenados nas fontes contendo 1.000.000 entradas no total, e a janela de eventos foi definida em 1.000 elementos. Nosso experimento consistiu em variar o número de instâncias dos estágios *filtra*, *persiste* e *correlaciona*, especificados

Tabela 5.2: Tempos totais de execução da aplicação de processamento complexo de eventos.

<i>Cenário</i>	<i>Threads</i>	<i>Número de Instâncias</i>			<i>Tempo total (s)</i>
		<i>Persiste</i>	<i>Filtra</i>	<i>Correlaciona</i>	
1	24	6	6	12	107,836
2	24	6	6	12	98,720
1	24	12	12	12	104,015
2	24	12	12	12	102,200
1	48	12	12	24	112,116
2	48	12	12	24	102,329
1	48	24	24	24	111,629
2	48	24	24	24	111,616

através de parâmetros de execução, assim como o tamanho do *pool* de *threads* em cada processo. Dois casos de teste foram executados em cada cenário, um utilizando 24 *threads* no total (uma por processador disponível) e 48 *threads* (duas por processador). No cenário com dois processos, o *pool* de *threads* foi dividido igualmente entre os processos. Diferentes configurações de instâncias para cada estágio foram experimentadas para verificar a configuração que resulta em melhor desempenho. A Tabela 5.2 contém os tempos totais, em segundos, de processamento de cada caso de teste para os dois cenários.

Um resultado interessante é que a divisão do processamento da aplicação em dois aglomerados geralmente apresenta melhores resultados do que a configuração em um único aglomerado. Esse resultado está relacionado ao fato de que parte dos recursos de processamento ficam exclusivamente alocados para o estágio de maior complexidade no cenário com dois aglomerados.

Os resultados demonstram que variações dos parâmetros de execução levam a diferenças de aproximadamente 13,5% no tempo de execução da aplicação entre o melhor e o pior caso, e, portanto, comprovam a importância da flexibilidade de experimentar com diferentes configurações em diferentes cenários de execução.

5.2 Sobrecarga da Implementação

Para avaliar a sobrecarga de processamento e dos mecanismos de comunicação da implementação, comparamos duas aplicações implementadas em Leda com outras soluções semelhantes consolidadas em ambientes de produção.

A primeira aplicação consiste na adaptação de um servidor HTTP de páginas estáticas e dinâmicas implementado puramente em Lua para o modelo de estágios. Este tipo de aplicação representa um exemplo tradicional de programação concorrente, apresentando grande carga de operações de entrada

e saída e, portanto, explora os mecanismos de entrada e saída assíncrona oferecidos pela nossa implementação.

A segunda aplicação implementa uma operação de ordenação de vetores em memória distribuída. Essa análise procura avaliar o custo de usar Leda como infraestrutura de comunicação em cenários de execução mestre-escravo com memória distribuída, utilizando os resultados de uma implementação que utiliza a biblioteca MPI (Pacheco1997) como mecanismo de comunicação.

5.2.1

Servidor HTTP de Páginas Estáticas e Dinâmicas

Para esta avaliação, nós adaptamos a implementação do servidor *web* Xavante (Xavante2004) em Lua para acomodar a utilização de estágios no tratamento de requisições do protocolo HTTP. Esse servidor é implementado inteiramente em Lua e tem como característica atender simultaneamente requisições por conteúdo estático e dinâmico através de co-rotinas. Nosso objetivo em utilizar estágios para tratar requisições HTTP é observar o desempenho dos mecanismos de comunicação intra-processo (pela passagem de descritores de arquivos e *sockets* entre estágios) e as facilidades de entrada e saída assíncrona oferecidas pela implementação de Leda.

O servidor Xavante, além de prover transferência de conteúdo estático, ou seja, transferência de conteúdos armazenado em arquivos, também suporta a geração de conteúdo dinâmico através de *scripts* Lua. Na implementação original, o Xavante cria uma co-rotina para atender cada requisição HTTP. As co-rotinas são responsáveis por receber os pedidos de clientes e despachar os arquivos requisitados ou por executar e enviar a saída de um *script* que gera um conteúdo dinâmico.

A adaptação da arquitetura do servidor Xavante para o modelo de estágios consistiu na divisão funcional do código executado por co-rotinas que atendem os clientes em estágios. A Figura 5.6 contém a estrutura geral da arquitetura do servidor HTTP em estágios implementado.

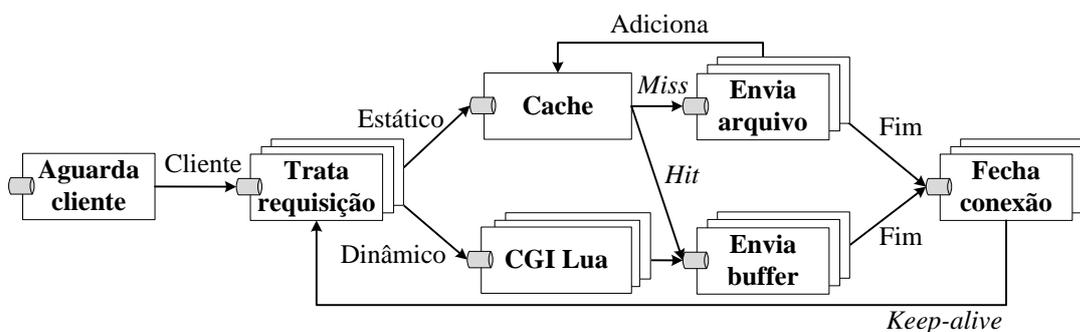


Figura 5.6: Estrutura do grafo de estágios do servidor HTTP em estágios.

O estágio *Aguarda cliente* executa um *loop* infinito que aguarda conexões TCP de clientes em uma porta pré-definida e as passa adiante. A implementação de Leda provê interfaces assíncronas para *sockets*, portanto esse estágio não bloqueia *threads* de sistema operacional enquanto aguarda por novas conexões.

O estágio *Trata requisição* é responsável por ler e processar as requisições de clientes. O resultado desse processamento é uma tabela Lua que descreve os cabeçalhos HTTP da requisição. Após o processamento, o estágio determina o tipo de conteúdo requisitado (dinâmico ou estático, dependendo do arquivo requisitado) e envia a tabela contendo a requisição do cliente, junto ao *socket* do cliente, para a porta de saída apropriada.

No caso de páginas estáticas, a requisição passa pelo estágio *cache* que possui estado interno persistente e armazena em memória arquivos lidos previamente pelo servidor. O estágio *Envia arquivo* trata eventos da porta *miss* da *cache* e usa as interfaces de entrada e saída assíncrona para ler arquivos do disco e enviar os cabeçalhos e pacotes lidos para clientes. Um evento de conclusão da leitura do arquivo é passado de volta para a inserção dos dados lidos na *cache*. A *cache* implementada usa uma política simples de gerenciamento de objetos armazenados baseada na política LRU (*Least Recently Used*) e possui um tamanho máximo configurável.

Requisições a páginas dinâmicas são tratadas pelo estágio *CGI Lua*, que é responsável por executar *scripts* Lua e enviar o conteúdo dinâmico adiante. O estágio *Envia buffer* envia dados em memória dos estágios *cache* e *CGI Lua* para o *socket* do cliente.

Caso o cliente tenha requisitado uma conexão persistente, o *socket* do cliente é repassado ao estágio *Trata Requisição* para iniciar o processo de leitura de uma nova requisição. Caso contrário, a conexão é fechada e o atendimento ao cliente se encerra.

Com exceção do estágio *cache*, estágios do servidor não necessitam de estado persistente devido à natureza sem estados do protocolo HTTP e, portanto, clientes são atendidos por instâncias paralelas. Todos os conectores usados para interligar os estágios da aplicação são locais, uma vez que os estágios precisam trocar descritores de *socket*, e portanto, devem executar em um único processo de sistema operacional.

Realizamos testes para determinar o comportamento da implementação do servidor HTTP em estágios, comparando-a com duas implementações de servidores HTTP usados em ambientes de produção: Apache (Apache2013) (versão 2.2.22), que possui uma arquitetura baseada em múltiplas *threads*, e Nginx (Nginx2013) (versão 1.5.2), que possui uma arquitetura orientada

a eventos, ambos implementados em C. O servidor Apache foi configurado para pré-inicializar 150 *threads* e criar um número máximo de 2000 *threads* durante sua execução. Os servidores Leda e Nginx foram configurados para criar 24 *threads* (ou processos no caso do Nginx) de sistema operacional para atendimento de clientes, um por núcleo disponível.

Utilizamos como servidor uma máquina do *cluster* exclusiva para cada versão do servidor HTTP testado. Usamos dez máquinas adicionais do *cluster* para gerar requisições aos servidores. Como aplicação cliente, utilizamos a ferramenta HTTPPerf (HTTPPerf2013) versão 0.9.0, que realiza diversas requisições HTTP concorrentes de acordo com uma configuração e exibe um relatório final contendo a média dos tempos de resposta, vazão média de requisições atendidas e erros ocorridos.

Os testes consistiram em configurar os clientes para abrir uma quantidade predefinida de conexões para o servidor HTTP e efetuar 100 requisições por conexão. Nós variamos entre 10 e 500 o número de conexões concorrentes em cada cliente, que totalizam, no máximo, 5000 conexões simultâneas nos servidores em cada teste.

Os casos de teste trataram três situações: provisão de conteúdo estático de arquivos pequenos, provisão de conteúdo estático de arquivos grandes e provisão de conteúdo dinâmico.

O primeiro caso de teste consistiu em requisitar arquivos de 300 *kilobytes* com o objetivo de analisar o comportamento de servidores de acordo com um número crescente de requisições de curta duração. A Figura 5.7 mostra o resultado dos testes com relação à vazão média de requisições atendidas com sucesso por segundo em cada servidor.

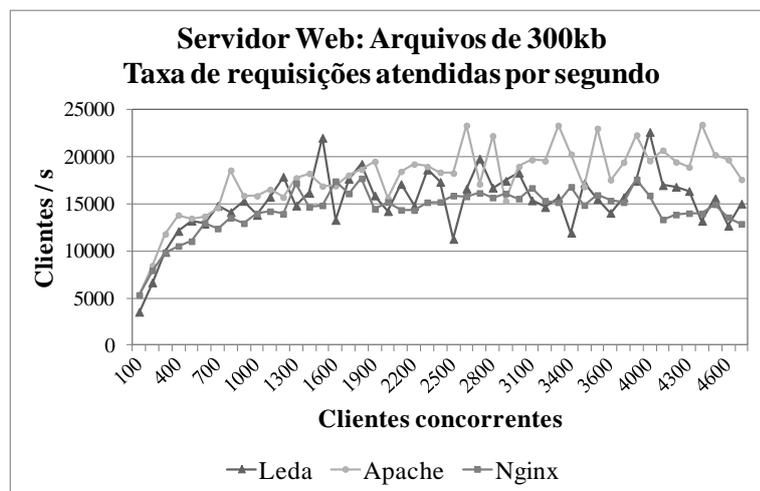


Figura 5.7: Comparação da vazão de requisições atendidas dos servidores *web* com páginas estáticas e arquivos de 300kb.

Podemos ver no gráfico da Figura 5.7 que, nesse cenário, o Apache alcança melhor desempenho entre as implementações de servidores testadas. Aparentemente o limite imposto pela banda passante da rede é alcançado aproximadamente em 700 clientes concorrentes e cria um gargalo para os servidores, fazendo com que a vazão de atendimento de clientes observada se estabilize após esse limite.

A Figura 5.8 mostra o tempo médio de atendimento de requisições atendidas em cada servidor para arquivos de 300 *kilobytes*. Essa métrica é definida pelo tempo total que um servidor leva para atender uma requisição, incluindo o tempo de um eventual estabelecimento de conexão TCP.

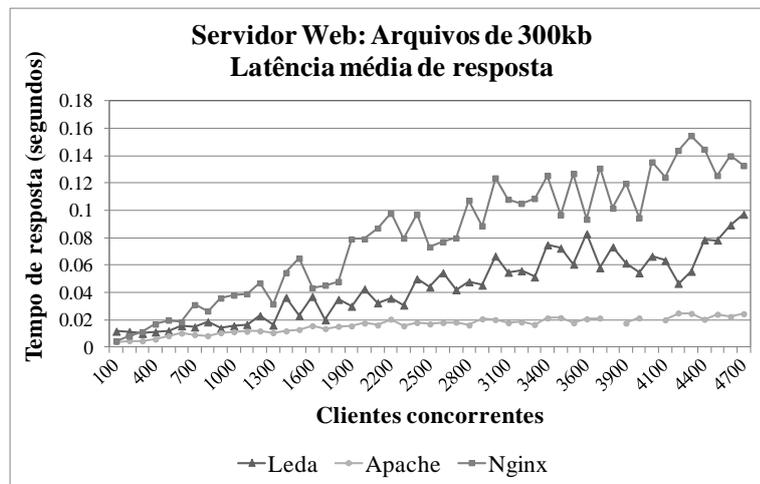


Figura 5.8: Comparação da latência média de resposta dos servidores *web* com páginas estáticas e arquivos de 300kb.

O tempo de atendimento do servidor Apache se manteve o menor, pois sua arquitetura de múltiplas *threads* trabalhadoras é mais adequada para manter a latência baixa em situações em que o servidor não está sobrecarregado. O fato de todas as requisições terem sido atendidas com sucesso (sem erros de tempo limite) corrobora essa conclusão.

Podemos ver que as implementações Leda e Nginx se comportam como sistemas orientados a eventos, em que a latência aumenta linearmente de acordo com o número de conexões simultâneas devido ao uso de filas de eventos. Porém, a implementação Leda obteve melhores resultados nesse caso de teste. Esse comportamento está relacionado ao fato de que o custo de troca de contexto entre processos Nginx é maior do que o tempo de troca de contexto de *threads* dentro de um processo Leda.

Em seguida, aumentamos o tamanho dos arquivos requisitados até sobrecarregar o sistema de arquivos, fazendo com que as requisições comecem a falhar devido ao tempo limite máximo de conexão ser alcançado. A Figura 5.9

mostra o resultado dos testes com relação a vazão média de requisições atendidas com sucesso com requisição à arquivos de 5 *megabytes*.

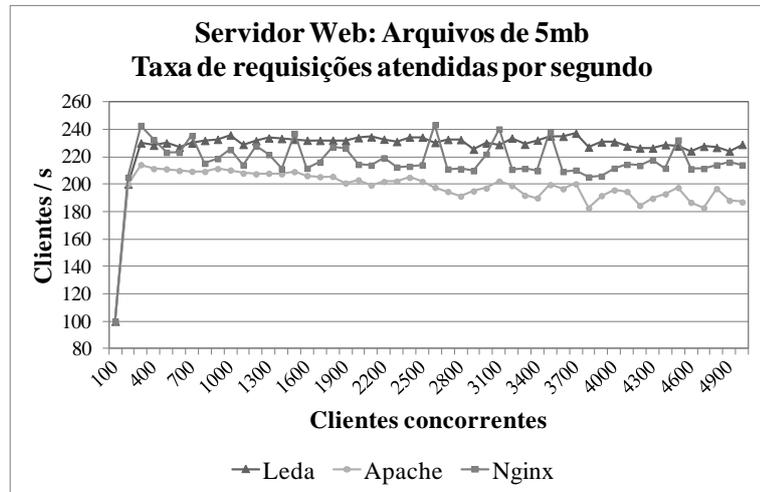


Figura 5.9: Comparação da vazão de requisições atendidas dos servidores *web* com páginas estáticas e arquivos de 5mb.

A taxa média de atendimento de clientes nesse caso atinge seu patamar entre 200 e 300 clientes simultâneos. Nesse caso de teste, o Apache apresenta uma degradação na taxa de requisição atendidas por segundo quando sobrecarregado devido a utilização de um grande número de *threads* de sistema operacional. As implementações Leda e Nginx mantiveram uma vazão constante de requisições atendidas, com Leda apresentando resultados um pouco melhores. Esse comportamento é esperado em sistemas orientados a eventos, em que a vazão de eventos permanece constante quando a aplicação está sobrecarregada e o excesso de clientes é acumulado nas filas de eventos.

A Figura 5.10 mostra o tempo médio de atendimento de requisições em cada servidor para arquivos de 5 *megabytes*.

Nesse cenário de teste, o tempo médio de atendimento de requisições do servidor Apache aumenta de forma repentina quando o número de clientes simultâneos ultrapassa de 2.000. Nesse ponto, o número máximo de *threads* trabalhadoras definidas para o servidor é ultrapassado, causando grande degradação de latência de atendimento. Os servidores Leda e Nginx, quando sobrecarregados, degradaram a latência de forma mais suave. O Nginx manteve um tempo médio menor de atendimento de requisições com relação a implementação Leda. No servidor Leda, notamos um aumento de erros de tempo limite de forma proporcional ao número de clientes simultâneos, porém, a latência nesse caso se mostrou constante.

O último cenário de teste, de conteúdo dinâmico, consiste em solicitar um *script* Lua que executa um laço *for* de 30.000 iterações, gerando uma

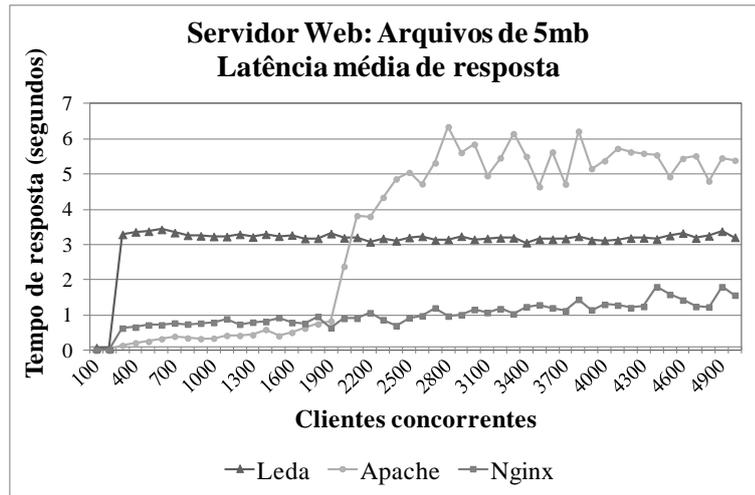


Figura 5.10: Comparação da latência média de resposta dos servidores *web* com páginas estáticas e arquivos de 5mb.

página HTML de 319 *kilobytes* contendo uma sequência de números. Esse caso é interessante para verificar o comportamento dos servidores em uma situação com maior demanda de processamento. A seção B.1 contém o código do *script* usado nos testes.

Nesse cenário, configuramos o Apache para tratar páginas dinâmicas Lua através do módulo *mod_lua*, que permite a utilização do mesmo componente *CGI Lua* usado pelo xavante em sua execução. Assim, usamos o mesmo *script* para gerar as páginas dinâmicas em ambos os casos para comparar os resultados da execução dos testes.

A Figura 5.11 contém o gráfico com os resultados dos testes com relação a vazão média de requisições atendidas ao *script* Lua dinâmico.

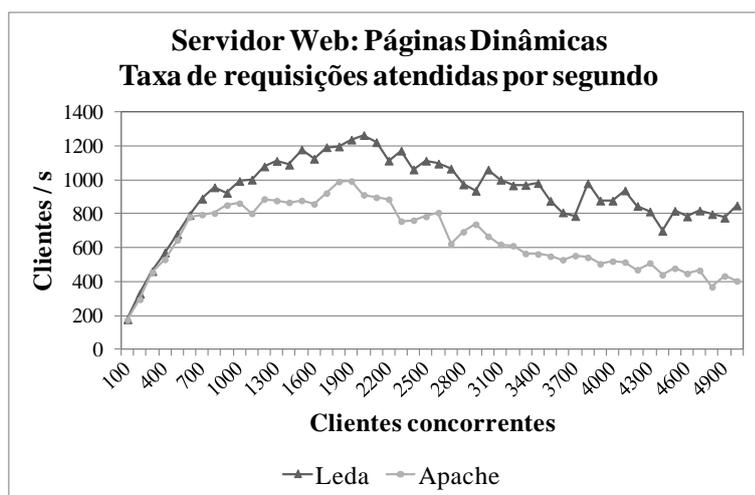


Figura 5.11: Comparação da vazão de requisições atendidas dos servidores *web* com páginas dinâmicas.

Nesse caso de teste, a taxa de atendimento de clientes por segundo se

mantêm a mesma para os dois servidores até aproximadamente 600 clientes simultâneos. A partir desse número, a implementação Leda alcança uma vazão de requisições maior com relação ao Apache.

Esse comportamento está relacionado à carga de processamento elevada de processamento do *script* Lua para gerar as páginas dinâmicas, que gera uma sobrecarga de troca de contexto grande para o modelo com muitas *threads* seguido pelo Apache. A implementação Leda utiliza uma *thread* por núcleo disponível, possuindo, portanto, uma sobrecarga menor de gerenciamento de *threads* pelo sistema operacional.

A Figura 5.12 contém os tempos médios de atendimento de requisições pelo *script* Lua dinâmico.

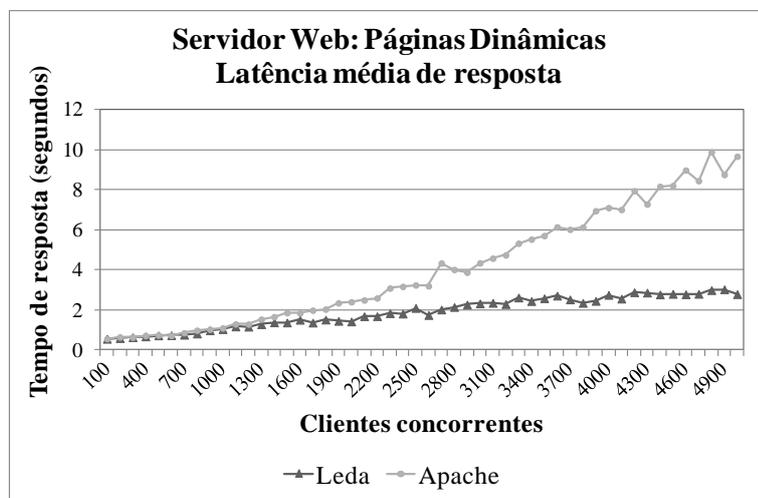


Figura 5.12: Comparação da latência média de resposta dos servidores *web* com páginas dinâmicas.

Nessa avaliação, podemos ver o efeito da sobrecarga da utilização de concorrência baseada em *threads* pelo Apache quando a requisição envolve uma carga de processamento grande. A curva de latência da implementação Leda cresce de forma mais suave e exibe comportamento linear pela utilização mais eficiente dos recursos de processamento disponíveis no servidor, além do uso de interfaces de entrada e saída assíncrona para *sockets*.

Os resultados dos testes confirmam que esse tipo de aplicação se beneficia da estratégia de mapeamento $N \times M$ de tarefas para *threads* de sistema operacional. A implementação do servidor HTTP em Leda permite que requisições sejam tratadas dentro da aplicação em paralelo com o uso de mecanismos de entrada e saída assíncronos, onde as requisições são tratadas de forma concorrente a medida em que transitam entre os estágios.

5.2.2 Ordenação de Vetores em Memória Distribuída

Nessa aplicação teste iremos avaliar o custo de comunicação de Leda imposta pela troca de eventos entre estágios que executam em ambientes distribuídos. O objetivo desse caso de teste é comparar a carga de computação pelo uso de Leda em diferentes níveis de granularidade de computação e comunicação com uma implementação em C que usa a biblioteca MPI.

Para isso, nós implementamos uma aplicação mestre-escravo para ordenação paralela de vetores. O algoritmo implementado opera da seguinte forma: primeiro inicializamos um vetor v com n elementos, que é dividido em p partes com $n \div p$ elementos sequenciais por um estágio divisor, que as envia para p portas de saída, cada uma conectada a um estágio que é mapeado a um processo remoto.

Esses estágios executam uma função implementada em C que ordena o vetor recebido. O resultado dessa ordenação é repassado para um estágio redutor após o retorno da execução dessa função. O algoritmo termina após o estágio redutor receber todas as p partes ordenadas do vetor. A Figura 5.13 contém o grafo de estágios implementado para essa aplicação.

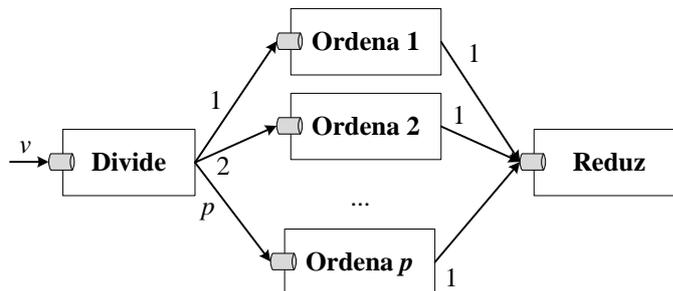


Figura 5.13: Estrutura do grafo de estágios da aplicação distribuída de ordenação de vetores.

A ordenação paralela de vetores é um problema que tem sido amplamente abordado nos últimos anos para uma variedade de arquiteturas de computadores. Está além do escopo deste estudo elaborar um algoritmo de ordenação paralelo eficaz, portanto utilizamos o algoritmo *selection sort* (Leiserson2001) pelo fato de sua complexidade de pior e melhor caso variar apenas de acordo com o tamanho do vetor a ser ordenado.

Iremos comparar o custo de comunicação de nossa aplicação com uma versão desse algoritmo implementada em C, que usa a biblioteca MPI para particionar e transmitir o vetor para processos remotos e transmitir os resultados parciais de volta ao processo de origem. Para compilar e executar a aplicação MPI, usamos a implementação da biblioteca MPI da Intel em sua versão 4.0.1.

Tabela 5.3: Comparação dos tempos de execução de Leda e MPI para aplicação distribuída de ordenação de vetores.

<i>Tamanho</i>	<i>Processos</i>	<i>Tempo Leda (s)</i>	<i>Tempo MPI (s)</i>	<i>Diferença</i>
1536	4	0,008	0,001	1057%
	8	0,012	0,002	415%
	16	0,022	0,004	357%
	24	0,031	0,008	282%
	32	0,039	0,001	1973%
192000	4	4,418	4,162	6%
	8	1,488	1,051	42%
	16	0,362	0,267	35%
	24	0,220	0,123	78%
	32	0,173	0,085	102%
384000	4	17,520	16,632	5%
	8	4,487	4,159	8%
	16	1,825	1,044	75%
	24	0,905	0,470	93%
	32	0,532	0,268	98%
768000	4	69,811	66,456	5%
	8	17,618	16,661	6%
	16	4,609	4,169	11%
	24	2,174	1,861	17%
	32	1,609	1,067	50%
1536000	4	278,951	266,356	5%
	8	70,022	66,505	5%
	16	17,815	16,647	7%
	24	8,182	7,417	10%
	32	4,810	4,180	15%

A divisão do vetor e a comunicação é realizada através da função `MPI_Scatter` do MPI e cada processo MPI executa a mesma função C de ordenação de vetores usada na versão da aplicação implementada em Leda. Os códigos fonte das aplicações estão listados no Anexo B desse trabalho.

A medição do tempo de execução dessa aplicação foi realizada através da função `gettimeofday` da API C Posix, que fornece a hora atual e possui resolução em microssegundos. Nós desconsideramos os tempos de inicialização do vetor e de carregamento das bibliotecas e processos externos. Todos os testes foram executados dez vezes e os resultados mostrados abaixo representam a média aritmética das medições realizadas.

A Tabela 5.3 contém uma comparação dos resultados dos tempos de execução das duas versões da aplicação em cenários de execução utilizando diferentes quantidades de processos remotos e tamanhos do vetor a ser ordenado. Nos dois casos de teste (Leda e MPI), cada processo executa em um nó exclusivo do *cluster*.

Tabela 5.4: Tempos de comunicação de Leda e MPI entre processos remotos.

<i>Tamanho do vetor (em bytes)</i>	<i>Tempo Leda (s)</i>	<i>Tempo MPI (s)</i>
12288	0,0002	0,00001
1536000	0,0113	0,0008
3072000	0,0260	0,0015
6144000	0,0547	0,0033
12288000	0,1109	0,0079

A primeira coluna da Tabela 5.3 contém o tamanho do vetor em elementos, usado em cada caso de teste. Cada elemento do vetor é um valor do tipo *double* (8 bytes). A segunda coluna contém a quantidade de processos externos que executam a operação de ordenação do vetor. A terceira coluna mostra o tempo de execução da versão Leda do algoritmo e a quarta coluna possui o tempo da versão MPI. Os tempos de execução são apresentados em segundos. Finalmente, a última coluna contém a razão dos tempos de execução dos dois algoritmos para fins de comparação.

Um aspecto interessante que é observado nos resultados é que a razão dos tempos de execução aumenta à medida que utilizamos mais processos paralelos. Isso fica mais evidente conforme o tamanho do vetor diminui, pois o custo para tratar um conjunto de entrada menor se torna mais alto em uma linguagem interpretada, como Lua.

Para verificar puramente o custo de comunicação, nós realizamos a medição do tempo de envio de um vetor para apenas um processo remoto. A Tabela 5.4 mostra os tempos de envio medidos.

Os resultados demonstram que a implementação de Leda possui uma sobrecarga de comunicação maior entre estágios mapeados a processos distribuídos que introduz um custo adicional com relação a implementação da biblioteca MPI utilizada. Esse resultado é consequência da necessidade de serialização do vetor antes da transmissão, que é desnecessário em MPI, uma vez que os dados são transmitidos diretamente do *buffer* de memória em que o vetor está alocado. Além disso, MPI aplica otimizações na implementação de comunicação entre processos.

5.3

Facilidade do Uso de Leda

Para avaliar a facilidade do uso de Leda, descreveremos o processo de implementação de duas aplicações não convencionais: uma aplicação para monitoração contínua do valor de ações do pregão da bolsa de valores Bovespa, que avalia as facilidades de reuso e composição de estágios, e uma aplicação para simulação de tubos ascendentes de petróleo, implementada como experi-

mento da aplicabilidade de Leda em problemas que lidam com o processamento de arquivos com grandes volumes de dados.

5.3.1 Visualização de Ações da Bolsa de Valores

Uma das características do modelo de programação em etapas proposto é prover facilidades para reuso da especificação de um estágio em diversos contextos de aplicações. Assim, nesta avaliação, são analisadas as questões referentes ao reuso de estágios e sua composição em uma aplicação de maior nível de abstração.

Para tal, desenvolvemos um programa para monitoração contínua do valor de ações da bolsa de valores através da recuperação periódica dos valores de uma ação específica do sítio do pregão *online* da Bovespa².

O código fonte completo da aplicação é mostrado na Listagem 5.1 e consiste em um *script* Lua contendo 23 linhas de código, que utiliza apenas estágios que são fornecidos com a instalação padrão da biblioteca Leda através de módulos extras.

Listagem 5.1: Código-fonte da aplicação de Visualização de preços de ações da Bolsa de Valores.

```

1 local leda=require 'leda '
2 local timer=require 'leda.stage.util.timer '
3 local http=require 'leda.stage.net.http '
4 local match=require 'leda.stage.string.match '
5 local window=require 'leda.stage.window.sliding '
6 local map=require 'leda.stage.util.map '(function(v)
7     return string.gsub(v, ', ', '. ')
8 end)
9 local media=require 'leda.stage.util.average '
10 local printer=require 'leda.stage.util.print '
11 local g=leda.graph{
12     leda.connect(timer, http),
13     leda.connect(http, match),
14     leda.connect(match, window),
15     leda.connect(window, map),
16     leda.connect(map, media),
17     leda.connect(media, printer)
18 }
19 match.pattern='<td>#. -</td>%s<td>. -(.-)</td>'
20 window.size=5
21 local url=assert(arg[1], "URL requerida ")
22 timer:push(1, -1, url)
23 g:run()

```

²<http://pregao-online.bmfbovespa.com.br>

O módulo `leda.stage.util.timer` (linha 2) retorna um estágio que implementa um temporizador, repassando os dados recebidos periodicamente. Esse estágio recebe como parâmetros o período de tempo entre eventos emitidos, o número de iterações (ou -1 para emitir eventos indefinidamente), e quaisquer parâmetros adicionais são repassados adiante em cada iteração. A partir do evento emitido na linha 22, o estágio irá emitir eventos a cada 1 segundo contendo uma *string* recebida como primeiro parâmetro do *script*.

O módulo `leda.stage.net.http` (linha 3) retorna um estágio que utiliza o protocolo HTTP para recuperar dados a partir de um endereço recebido como parâmetro. Os dados recebidos são emitidos diretamente para a porta padrão do estágio de índice 1 (porta padrão). Nesse exemplo, o estágio `http` irá receber o evento repassado pelo estágio `timer` periodicamente.

O módulo `leda.stage.string.match` (linha 4) retorna o estágio definido na Listagem 4.3 para casamento de padrões. O padrão definido na linha 19 foi construído especificamente para recuperar o valor atual de uma ação da página do sítio da Bovespa, armazenada na variável `url` (linha 21).

O módulo `leda.stage.window.sliding` (linha 5) retorna um estágio que implementa uma janela deslizante. Esse estágio armazena eventos recebidos em um vetor interno, requerendo estado persistente, e o repassa quando o tamanho da janela chega a um limite máximo, definido no campo *size* do estágio. Nesse exemplo, o tamanho definido é de 5 eventos (linha 20).

O módulo `leda.stage.util.map` (linha 6) retorna um estágio que recebe um vetor e aplica uma função passada como parâmetro para cada um de seus elementos. Nesse exemplo, a função passada altera o separador decimal dos valores armazenados no vetor repassado pelo estágio `window` de vírgula para a representação que utiliza ponto decimal, padrão utilizado por Lua para representar valores de ponto flutuante em *strings* numéricas.

O módulo `leda.stage.util.average` (linha 9) retorna um estágio que recebe um vetor contendo valores numéricos e emite um evento contendo a média aritmética desses valores.

O módulo `leda.stage.util.print` (linha 10) retorna o estágio definido na Listagem 4.1 para imprimir os valores emitidos pelo estágio `average`.

Portanto, a composição dos estágios definida pelo grafo entre as linhas 11 e 18 resulta em uma aplicação que recebe o endereço da página de uma ação específica do sítio da Bovespa como parâmetro e imprime indefinidamente em sua saída padrão o valor da média das últimas cinco medições realizadas a cada segundo.

Essa aplicação demonstra que é possível construir programas complexos

a partir do reuso e composição de estágios desenvolvidos separadamente. Diferentemente de SEDA, os estágios foram reutilizados sem a necessidade de alterar a especificação dos módulos usados.

5.3.2

Simulação de Tubos Ascendentes de Petróleo

Em trabalhos anteriores (Salmito2013) nós descrevemos como Leda pôde ser utilizado na implementação de um programa que processa arquivos com grandes volumes de dados para o laboratório Tecgraf da PUC-Rio.

Esse laboratório desenvolve um projeto com a companhia petrolífera brasileira (Petrobrás) para estudar, simular e visualizar a quebra de tubos ascendentes de petróleo em plataformas marítimas. O arquivo de entrada para a visualização contém a geometria inicial dos pontos que compõem o tubo de petróleo e a posição dos pontos em cada passo da simulação. Essa informação é processada através de um método de visualização de elementos finitos. Nesse caso, os elementos denotam cilindros aproximados por prismas com faces retangulares entre cada ponto da simulação. Quanto maior for o número de faces e a quantidade de pontos, melhor será a qualidade da visualização. A saída do método dos elementos finitos é renderizada na tela do computador. Geralmente todo o processo é repetido várias vezes para se obter melhores resultados.

A equipe desse projeto precisava desenvolver um programa para manipular arquivos com um grande número de passos. Para pequenos conjuntos de dados, depois que o arquivo é lido e o método dos elementos finitos é aplicado, os dados resultantes são mantidos na memória e renderizados continuamente em cada passo da simulação. No entanto, para um grande número de passos, não é viável para manter todos os dados de visualização em memória.

A biblioteca Leda foi utilizada para solucionar esse problema da seguinte forma: um estágio inicial (*reader*) lê o arquivo de entrada e encaminha os dados para o estágio que implementa o método de prismas finitos através de um algoritmo de extrusão (*extrusion*). Esse estágio pode se beneficiar da decomposição em domínio, uma vez que diferentes prismas podem ser calculados de forma independente. Os resultados da extrusão são encaminhados para um último estágio com estado persistente (*renderer*) que atua como um redutor, acumulando os prismas para a renderização. Esse último estágio pode receber dados relacionados aos diferentes passos de eventos de forma intercalada. Ele mantém uma tabela Lua contendo uma entrada para cada passo da simulação. Ao receber todos os dados relativos ao próximo passo, este é marcado como pronto e é exibido por um tempo mínimo na tela. A

Listagem 5.2 contém o *script* de configuração do grafo da aplicação.

Listagem 5.2: Um exemplo simplificado da definição do grafo da aplicação.

```

1 local reader = leda.stage{
2   handler = do_read ,
3   init = init_read ,
4   autostart = true
5 }
6 local extrusion = leda.stage{
7   handler = do_extrusion ,
8   init = init_extrusion
9 }
10 local renderer = leda.stage{
11   handler = do_render ,
12   init = init_render ,
13   serial = true
14 }
15 graph = leda.graph{
16   reader:connect( "segment " , extrusion ) ,
17   extrusion:connect( "prism " , renderer )
18 }
19 — Exemplo de partição do grafo
20 graph:part( graph:all() - extrusion , extrusion )
21 graph:map( "host1:port " , "host2:port " )
22 graph:run()

```

As linhas 1, 6 e 10 criam os três estágios da aplicação. Como dito anteriormente, a função `leda.stage` cria novos estágios, recebendo uma tabela Lua contendo a definição dos tratadores de eventos (nesse caso, as funções `do_read`, `do_extrusion` e `do_render`), e opcionalmente uma função de inicialização (respectivamente `init_read`, `init_extrusion` e `init_render`). A implementação dessas funções são mostradas na Listagem 5.3.

Listagem 5.3: Exemplo dos tratadores de eventos da aplicação da Listagem 5.2 (os códigos das funções auxiliares foram omitidos).

```

1 local input_file = assert( arg[1] )
2 local sides = assert( arg[2] )
3 function init_read()
4   require io
5   function read_segment( fd )
6     ... — Código omitido
7   end
8 end
9 function do_read( )
10  local fd = io.open( input_file , "r " )
11  for line_segment in read_segment( fd ) do
12    leda.send( "segment " , line_segment )

```

```

13  end
14  fd:close()
15 end
16 function init_extrusion()
17     function calc_prism(segment)
18         ... — Código omitido
19     end
20 end
21 function do_extrusion(segment)
22     local line_element = calc_prism(segment, sides)
23     leda.send("prism", line_element)
24 end
25 function init_render()
26     require "table"
27     lines = {}
28     current_line = 1
29     maxlines = ...
30 end
31 function do_render(element)
32     if not lines[element.line] then
33         lines[element.line] = {}
34     end
35     table.insert(lines[element.line], element)
36     if complete(current_line) then
37         ... — renderiza linha atual (Código omitido)
38         lines[current_line] = nil
39         current_line = (current_line+1) % maxlines
40     end
41 end

```

O parâmetro de configuração *autostart*, usado na criação do estágio *reader* (linha 4 da Listagem 5.2), indica que uma instância para esse estágio deve ser criada automaticamente e posta na fila de prontos do processo. O parâmetro *serial* indica que o estágio *renderer* deve ter uma única instância com estado persistente. O estágio *extrusion* possui instâncias com estado transientes e, portanto, podem possuir diversas instâncias executando em paralelo durante a execução.

A linha 15 da Listagem 5.2 invoca a função `leda.graph` para configurar o grafo de estágios da aplicação. Neste ponto, o programador cria conectores para ligar as portas de saída na fila de eventos do estágio de destino. Durante a execução, quando uma instância de estágio emite um evento para uma porta de saída, o conector correspondente irá direcioná-lo para a fila de eventos (local ou remota) correspondente.

O método `part`, chamado na linha 20, particiona o grafo da aplicação em

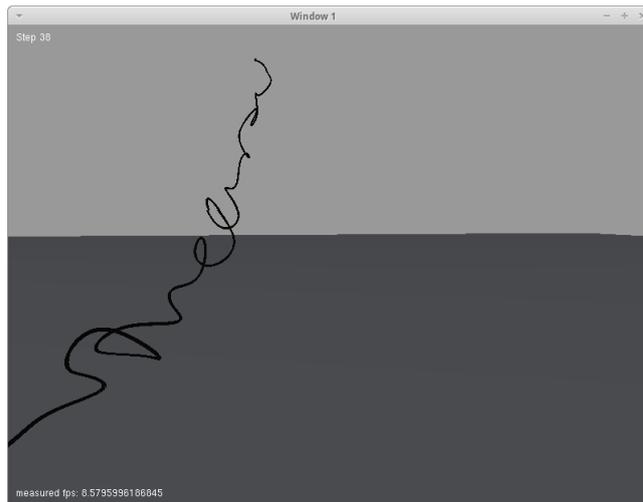


Figura 5.14: Captura de tela da aplicação de simulação de tubos ascendentes de petróleo.

aglomerados. Nesse exemplo, a aplicação é particionada em dois aglomerados: um contendo um único estágio (*extrusion*) e o outro contendo o resto da aplicação.

A linha 21 chama o método `map` para vincular os aglomerados a processos de sistema operacional que aguardam em uma porta TCP.

Finalmente, na linha 22, a invocação do método `run` inicia a execução da aplicação. A Figura 5.14 exibe a saída do estágio de renderização do programa.

Na versão original do programa, como já mencionado, os dados de renderização de todos os passos da simulação é mantida na memória. Na versão Leda, logo que o estágio de renderização move-se para um novo passo, os dados do último passo são descartados. Isso é trivial em Leda devido ao coletor de lixo de Lua.

A experiência foi considerada positiva pelos usuários, que agora podem trabalhar com conjuntos muito grandes de dados em cenários reais de operação de plataformas de petróleo.

6

Conclusão

Modelos de concorrência híbridos constituem uma importante tendência no desenvolvimento de aplicações e serviços que precisam lidar com um grande número de clientes simultâneos ou processar um grande conjunto de dados. A combinação de *threads* preemptivas e eventos cooperativos em um mesmo ambiente de execução permite que aplicações explorem o modelo mais apropriado para cada tipo de problema.

Outros trabalhos investigaram alternativas para reduzir a complexidade da combinação de *threads* e eventos em um ambiente de execução híbrido. Schimdt *et al.* descrevem um conjunto de modelos de concorrência híbridos em seu livro sobre padrões de programação concorrente (Schmidt2000), porém sem classificá-los. Pai *et al.* proporam uma classificação de modelos de concorrência para o caso específico de servidores *web* (Pai1999). Eles identificaram quatro tipos de servidores, nos quais alguns podem ser considerados classes de concorrência híbrida. No entanto, o foco do trabalho não é a combinação de modelos de concorrência, mas a comparação da arquitetura proposta com implementações alternativas de servidores *web*.

A primeira contribuição deste trabalho foi aprofundar a discussão sobre como lidar com a combinação de *threads* e eventos em um ambiente de execução híbrido. Para suprir a ausência de uma definição para o conceito de concorrência híbrida propusemos um sistema de classificação que se baseia na abstração que é oferecida ao programador – *threads* ou eventos – e a possibilidade de execução paralela em múltiplos processadores.

Entre as propostas existentes para modelos de concorrência híbridos, a arquitetura em estágios destaca-se por permitir que o programador equilibre o uso de *threads* e eventos, explorando ambas as abstrações. Além disso, o modelo de estágios separa a execução de partes da aplicação, permitindo que as decisões de escalonamento sejam adaptadas às características de cada estágio.

Uma série de extensões foram propostas para o modelo de estágios original (Li2006, Upadhyaya2007, Han2009, Gordon2010). Algumas dessas propostas tratam a limitação de comunicação em memória compartilhada usando diferentes implementações de filas de eventos, que podem conectar estágios lo-

calizados em diferentes processos de sistema operacional executando em ambientes distribuídos (Upadhyaya2007, Han2009). Porém, essas abordagens normalmente não adotam soluções específicas para o problema de condições de corrida em memória compartilhada dentro de estágios, e exigem que técnicas de sincronização de acesso ao estado compartilhado sejam aplicadas em tratadores de eventos. Esses trabalhos também não tratam o desacoplamento da especificação de estágios, dificultando o reuso de estágios em aplicações diferentes.

O modelo proposto neste trabalho estende a flexibilidade provida pelo modelo de estágios para oferecer um processo de desenvolvimento em etapas que desacopla as atividades relacionadas à especificação da lógica de funcionamento de aplicações das decisões relacionadas ao ambiente de execução. O modelo proposto permite que estágios sejam agrupados em diferentes configurações de aglomerados de acordo com as necessidades de escalonamento de tarefas e a granularidade de processamento de partes específicas da aplicação. Uma vez que a etapa de aglomeração de estágio e o mapeamento a unidades de execução são separados da definição da aplicação, o programador pode explorar diversas configurações para determinar a que reflete melhores resultados para a aplicação no ambiente de execução em questão.

Outra contribuição importante deste trabalho foi a incorporação do processo PCAM para desenvolvimento de aplicações concorrentes ao modelo de estágios para flexibilizar as estruturas de comunicação e acomodar ambientes de execução diferentes em uma mesma especificação de uma aplicação.

A biblioteca Leda, implementada no contexto desse trabalho, explora as características da linguagem de programação Lua para oferecer um ambiente de execução flexível, que permite que estágios compartilhem múltiplas *threads* de sistema operacional de forma cooperativa e que partes de aplicações concorrentes sejam desenvolvidas de forma isolada e independente.

A implementação de Leda foi validada através da implementação de um conjunto de aplicações que exploram a flexibilidade provida pelo modelo proposto de diferentes formas. Além dos exemplos de aplicação apresentados neste trabalho, outras aplicações e estágios reusáveis foram implementados para resolver problemas que necessitam dos mecanismos flexíveis oferecidos por Leda.

6.1 Sumário de Contribuições

Ao desenvolver este trabalho, buscamos alcançar um entendimento adequado do conceito de concorrência híbrida e da flexibilidade do processo de

desenvolvimento orientado a estágios. Nossas principais contribuições são:

- A investigação e a classificação de soluções que implementam concorrência híbrida, permitindo um entendimento mais profundo das limitações e vantagens das classes de soluções levantadas.
- Um modelo de concorrência baseado na arquitetura em estágios para explorar o desacoplamento da especificação de uma aplicação e seu mapeamento para diferentes cenários de execução de forma flexível.
- A comprovação, através de uma plataforma implementada em Lua e de um conjunto de experimentos que exploram sua flexibilidade, da efetividade desse modelo de concorrência.

6.2

Trabalhos Futuros

A pesquisa realizada ao longo deste trabalho e os resultados obtidos pela implementação do modelo proposto apontam novas direções de investigação.

Nesse aspecto, o passo inicial seria aprofundar o estudo sobre a aplicabilidade de políticas de escalonamento e explorar técnicas dinâmicas que levam em consideração o estado de execução de um conjunto de estágios para controlar o uso de recursos em aglomerados de forma automática, como, por exemplo, políticas de escalonamento para estágios propostas em (Li2006, Gordon2010).

Também temos interesse em investigar o uso da plataforma desenvolvida neste trabalho em outros cenários de aplicação que se beneficiam dos mecanismos flexíveis oferecidos pelo modelo proposto.

Outra linha de trabalho consiste no estudo sobre a alteração da estrutura do grafo de estágios durante a execução da aplicação de forma similar a abordagem seguida por SEDA. Técnicas para determinar um particionamento ótimo do grafo de estágios e alterar configuração de mapeamento de processos da aplicação de forma proativa podem ser investigadas e aplicadas dinamicamente durante a execução da aplicação.

O ambiente de execução Leda simplifica o tratamento de questões relacionadas a distribuição de tarefas em processos localizados em máquinas distintas, assim o programador pode foca-se na semântica de funcionamento de aplicações em estágios. Outra linha de investigação nesse contexto é estender os mecanismos de comunicação para incluir requisitos de tolerância a falhas através de mecanismos para detectar processos falhos e a inclusão de novos processos durante a execução da aplicação.

Referências Bibliográficas

- [Adya2002] ADYA, A.; HOWELL, J.; THEIMER, M.; BOLOSKY, W. J. ; DOUCEUR, J. R.. **Cooperative Task Management Without Manual Stack Management**. In: PROC. GENERAL TRACK OF USENIX ANNUAL TECHNICAL CONFERENCE, p. 289–302, Monterey, CA, USA, 2002. 1, 2.1
- [Agha1985] AGHA, G.. **Actors: a model of concurrent computation in distributed systems**. MIT Press, Cambridge, MA, USA, 1986. 2.1.2
- [Apache2013] **Apache software foundation: The apache web server**. <http://www.apache.org>, 2013. Acessado em Julho de 2013. 5.2.1
- [Armstrong1996] ARMSTRONG, J.; VIRDING, R.; WIKSTRÖM, C. ; WILLIAMS, M.. **Concurrent Programming in ERLANG**. Prentice Hall, Hertfordshire, UK, 2nd edition, 1996. 2.2
- [Benton2004] BENTON, N.; CARDELLI, L. ; FOURNET, C.. **Modern concurrency abstractions for c#**. ACM Trans. Program. Lang. Syst., 26(5):769–804, Sept. 2004. 1
- [Bharti2005] BHARTI, S.; KAULGUD, V.; PADMANABHUNI, S.; DAS, A. S.; KRISHNAMOORTHY, V.; UNNI, N. K. ; OTHERS. **Fine grained seda architecture for service oriented network management systems**. International Journal of Web Services Practices, 1(1-2):158–166, 2005. 3.4
- [Dabek2002] DABEK, F.; ZELDOVICH, N.; KAASHOEK, F.; MAZIÈRES, D. ; MORRIS, R.. **Event-driven programming for robust software**. In: PROC. 10TH. WORKSHOP ON ACM SIGOPS EUROPEAN WORKSHOP, EW 10, p. 186–189, New York, NY, USA, 2002. 1, 2.1.1
- [Fahndrich2006] FÄHNDRICH, M.; AIKEN, M.; HAWBLITZEL, C.; HODSON, O.; HUNT, G.; LARUS, J. R. ; LEVI, S.. **Language support for fast and reliable message-based communication in singularity os**. In: ACM SIGOPS OPERATING SYSTEMS REVIEW, volumen 40, p. 177–190. ACM, 2006. 4.2

- [Foster1995] FOSTER, I.. **Designing and building parallel programs**, chapter 2. Addison-Wesley Reading, MA, 1995. 3.1
- [Garlan2010] GARLAN, D.; MONROE, R. ; WILE, D.. **Acme: An architecture description interchange language**. In: CASCON FIRST DECADE HIGH IMPACT PAPERS, p. 159–173. ACM, 2010. 3.1
- [Gaud2010] GAUD, F.; GENEVES, S.; LACHAIZE, R.; LEPEERS, B.; MOTTET, F.; MULLER, G. ; QUÉMA, V.. **Efficient workstealing for multicore event-driven systems**. In: DISTRIBUTED COMPUTING SYSTEMS (ICDCS), 2010 IEEE 30TH INTERNATIONAL CONFERENCE ON, p. 516–525. IEEE, 2010. 1
- [Ghemawat2004] GHEMAWAT, S.; DEAN, J.. **Mapreduce: Simplified data processing on large clusters**. In: PROC. 6TH. SYMPOSIUM ON OPERATING SYSTEM DESIGN AND IMPLEMENTATION (OSDI'04), SAN FRANCISCO, CA, USA, 2004. 2.2
- [Gordon2010] GORDON, M. E.. **Stage scheduling for CPU-intensive servers**. PhD thesis, University of Cambridge, Computer Laboratory, 2010. 1, 2.2, 3.4, 6, 6.2
- [Gribble2001] GRIBBLE, S.; WELSH, M.; VON BEHREN, R.; BREWER, E.; CULLER, D.; BORISOV, N.; CZERWINSKI, S.; GUMMADI, R.; HILL, J.; JOSEPH, A. ; OTHERS. **The ninja architecture for robust internet-scale systems and services**. *Computer Networks*, 35(4):473–497, 2001. 2.2
- [HTTPerf2013] **Hp software: Httpperf**. <http://www.hpl.hp.com/research/linux/httpperf>, 2013. Acessado em Julho de 2013. 5.2.1
- [Hakeem2010] AL HAKEEM, M. S.; HEISS, H.-U.. **Adaptive scheduling for staged applications: The case of multiple processing units**. In: INTERNET AND WEB APPLICATIONS AND SERVICES (ICIW), 2010 FIFTH INTERNATIONAL CONFERENCE ON, p. 51–60. IEEE, 2010. 3.4
- [Haller2007] HALLER, P.; ODESKY, M.. **Actors that unify threads and events**. In: PROC. 9TH. INTERNATIONAL CONFERENCE ON COORDINATION MODELS AND LANGUAGES, p. 171–190, Paphos, Cyprus, 2007. 1, 2.1.2
- [Haller2010] HALLER, P.; ODESKY, M.. **Capabilities for uniqueness and borrowing**. In: ECOOP 2010–OBJECT-ORIENTED PROGRAMMING, p. 354–378. Springer, 2010. 4.2

- [Han2009] HAN, B.; LUAN, Z.; ZHU, D.; REN, Y.; CHEN, T.; WANG, Y. ; WU, Z.. **An improved staged event driven architecture for Master-Worker network computing**. In: CYBER-ENABLED DISTRIBUTED COMPUTING AND KNOWLEDGE DISCOVERY. CYBERC '09, p. 184–190, Zhangjiajie, China, 2009. 2.1.3, 6
- [Haskell2013] **The haskell programming language**. <http://haskell.org>, 2013. Acessado em Julho de 2013. 2.1.2
- [IEEE/ANSI1996] IEEE/ANSI. **Std. 1003.1: Portable operating system interface (posix) – part 1: System application program interface (api)**. Technical Report Including 1003.1c: Amendment 2: Threads Extension C Language., ANSI/IEEE, 1996. 2.1.2
- [Ierusalimschy2006] IERUSALIMSCHY, R.. **Programming in Lua, Second Edition**. Lua.Org, 2006. 4, 4.3.1
- [Ierusalimschy2011] IERUSALIMSCHY, R.; DE FIGUEIREDO, L. ; CELES, W.. **Passing a language through the eye of a needle**. Commun. ACM, 54(7):38–43, July 2011. 4
- [Imlib2004] **Imlib2 library documentation**. <http://docs.enlightenment.org/api/imlib2/html/>, 2004. Acessado em Julho de 2013. 5.1.1
- [Killian2007] KILLIAN, C.; ANDERSON, J. W.; BRAUD, R.; JHALA, R. ; VAHDAT, A.. **Mace : Language Support for Building Distributed Systems**. In: PROC. ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, p. 179–188, San Diego, CA, USA, 2007. 2.1.1
- [Lea2000] LEA, D.. **Concurrent programming in Java: design principles and patterns**. Addison-Wesley Professional, 2nd edition, 2000. 1
- [Lee2006] LEE, E. A.. **The problem with threads**. Computer, 39(5):33–42, 2006. 1
- [Leiserson2001] LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. ; CORMEN, T. H.. **Introduction to algorithms**. The MIT press, 2001. 5.2.2
- [Li2006] LI, Z.; LEVY, D.; CHEN, S. ; ZIC, J.. **Auto-tune design and evaluation on staged event-driven architecture**. In: PROC. 1ST WORKSHOP ON MODEL DRIVEN DEVELOPMENT FOR MIDDLEWARE (MODDM'06), p. 1–6. ACM, 2006. 1, 3.4, 6, 6.2

- [Li2007] LI, P.; ZDANCEWIC, S.. **Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives**. In: PROC. 2007 ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, PLDI '07, p. 189–199, New York, NY, USA, 2007. 1, 2.1, 2.1.2
- [Mazieres2001] MAZIÈRES, D.. **A toolkit for user-level file systems**. In: PROC. USENIX TECHNICAL CONFERENCE, p. 261–274, Boston, MA, USA, 2001. 2.1.1
- [Moggi1991] MOGGI, E.. **Notions of computation and monads**. Information and computation, 93(1):55–92, 1991. 2.1.2
- [Nginx2013] **The nginx webserver**. <http://nginx.org>, 2013. Acessado em Julho de 2013. 5.2.1
- [Odersky2008] ODERSKY, M.; SPOON, L. ; VENNERS, B.. **Programming in Scala**. Artima Press, 2008. 2.1.2
- [Ousterhout1996] OUSTERHOUT, J.. **Why Threads Are A Bad Idea (for most purposes)**, Jan. 1996. 1, 2
- [Pacheco1997] PACHECO, P. S.. **Parallel programming with MPI**. Morgan Kaufmann Pub, 1997. 2.2, 5.2
- [Pai1999] PAI, V. S.; DRUSCHELY, P. ; ZWAENEPOELY, W.. **Flash: An efficient and portable web server**. In: PROC. USENIX ANNUAL TECHNICAL CONFERENCE, p. 15–15. USENIX Association, 1999. 2.1, 6
- [Papadopoulos1998] PAPADOPOULOS, G. A.; ARBAB, F.. **Coordination models and languages**. Advances in computers, 46:329–400, 1998. 3.1
- [Reinders2010] REINDERS, J.. **Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism**. O'Reilly Media, 2010. 4.3.2
- [Robbins2003] ROBBINS, K.; ROBBINS, S.. **Unix Systems Programming: Communication, Concurrency and Threads**. Prentice Hall PTR, 2003. 4.3.2
- [Russell2012] RUSSELL, J.; COHN, R.. **Libevent**. Book on Demand, 2012. 4.3.2

- [Salmito2013] SALMITO, T.; DE MOURA, A. L. ; RODRIGUEZ, N.. **A flexible approach to staged events**. In: PROC. 6TH. INTERNATIONAL WORKSHOP ON PARALLEL PROGRAMMING MODELS AND SYSTEMS SOFTWARE FOR HIGH-END COMPUTING. (Aceito para publicação), 2013. 5.3.2
- [Schmidt2000] SCHMIDT, D.; STAL, M.; ROHNERT, H. ; BUSCHMANN, F.. **Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects**, chapter 5. Wiley, 2000. 6
- [Silberschatz2009] SILBERSCHATZ, A.; GALVIN, P. B. ; GAGNE, G.. **Operating system concepts**. J. Wiley & Sons, 2009. 1
- [Silvestre2009] SILVESTRE, B. O.. **Modelos de Concorrência e Coordenação para o Desenvolvimento de Aplicações Orientadas a Eventos em Lua**. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro - PUC-Rio, 2009. 4.3.2
- [Silvestre2010] SILVESTRE, B.; ROSSETTO, S.; RODRIGUEZ, N. ; BRIOT, J.-P.. **Flexibility and coordination in event-based, loosely coupled, distributed systems**. *Comput. Lang. Syst. Struct.*, 36(2):142–157, 2010. 2.1.2
- [Skyrme2008] SKYRME, A.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **Exploring lua for concurrent programming**. *Journal of Universal Computer Science*, 14(21), 2008. 4.3.1
- [Sussman1998] SUSSMAN, G. J.; STEELE JR, G. L.. **Scheme: A interpreter for extended lambda calculus**. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998. 2.1
- [Toernig2000] TOERNIG, E.. **Coroutine library source**. <http://xmailserver.org/pcl.html>, 2000. Acessado em Julho de 2013. 2.1.2
- [Upadhyaya2007] UPADHYAYA, G.; PAI, V. S. ; MIDKIFF, S. P.. **Expressing and exploiting concurrency in networked applications with aspen**. In: PROC. 12TH. ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, PPOPP '07, p. 13–23, New York, NY, USA, 2007. 2.1.3, 2.2, 2.2, 3.3, 6
- [Ururahy2002] URURAHY, C.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **Alua: flexibility for parallel programming**. *Computer Languages, Systems & Structures*, 28:155–180, 2002. 2.2

- [Welsh2001] WELSH, M. D.; CULLER, D. ; BREWER, E.. **Seda: an architecture for well-conditioned, scalable internet services**. SIGOPS Operating Systems Rev., 35(5):230–243, 2001. 1, 2.1.3, 2.1.3, 2.1.3
- [Welsh2002] WELSH, M. D.. **An architecture for highly concurrent, well-conditioned internet services**. PhD thesis, University of California, 2002. 2.1, 2.1.3
- [Wu2006] WU, E.; DIAO, Y. ; RIZVI, S.. **High-performance complex event processing over streams**. In: PROCEEDINGS OF THE 2006 ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, p. 407–418. ACM, 2006. 5.1.2
- [Xavante2004] **Kepler project: Xavante webserver**. <http://www.keplerproject.org/xavante>, 2004. Acessado em Julho de 2013. 5.2.1
- [Yoo2011] YOO, S.; LEE, H.; KILLIAN, C. ; KULKARNI, M.. **Incontext: simple parallelism for distributed applications**. In: PROC. 20TH. INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, p. 97–108. ACM, 2011. 2.1.1
- [deMoura2009] DE MOURA, A. L.; IERUSALIMSKY, R.. **Revisiting coroutines**. ACM Transactions on Programming Languages and Systems (TOPLAS), 31(2), Feb. 2009. 2.1, 4
- [vanRossum1995] VAN ROSSUM, G.. **Python library reference**, chapter 7, p. 215–249. luniverse Inc, 1995. 1
- [vonBehren2003b] VON BEHREN, R.; CONDIT, J.; ZHOU, F. ; NECULA, G. C.. **Capriccio: Scalable threads for internet services**. In: PROC. 19TH. ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, p. 268–281, Bolton Landing, NY USA, 2003. 1, 2.1.2, 2.1.2, 2.2
- [vonBehren2003a] VON BEHREN, R.; CONDIT, J. ; BREWER, E.. **Why events are a bad idea (for high-concurrency servers)**. In: PROC. 9TH. CONFERENCE ON HOT TOPICS IN OPERATING SYSTEMS - VOLUME 9, p. 4–4, Lihue, HI, USA, 2003. 2

A

API Leda

Apresentamos aqui a descrição da API Leda para programação de aplicações em estágios. A seguir apresentamos primeiro as funções da API principal de Leda para criação de estágios, construção do grafo da aplicação e aglomeração e mapeamento em processos, além de funções para a criação de novos processos. Em seguida descrevemos as chamadas que podem ser feitas em instâncias de estágios.

A.1

A API Principal de Leda

Essa seção descreve as funções principais da API de Leda.

leda.stage({...})

Essa função cria e retorna um novo objeto de estágio que é definido por uma tabela Lua contendo os seguintes atributos:

‘**handler**’ tipo: **função** (atributo obrigatório): Essa é a função que será usada como o tratador de eventos do estágio criado. Essa função irá receber como parâmetro cada evento retirado da fila de eventos do estágio.

‘**init**’ tipo: **função** (atributo opcional): Caso seja definida, essa função será chamada após a criação de uma instância do estágio. Ela tipicamente faz a inicialização do estado interno da instância recém criada.

‘**bind**’ tipo: **função** (atributo opcional): Caso seja definida, essa função será chamada apenas uma vez antes da execução do grafo que contém esse estágio. Essa função recebe como parâmetro o objeto contendo a representação do estágio e uma tabela contendo as portas de saída definidas no grafo da aplicação. Essa função tipicamente faz verificações a respeito das conexões e atributos internos do estágio.

‘**serial**’ tipo: **booleano** (atributo opcional): Se o valor desse atributo for ‘**false**’ ou ‘**nil**’, o estágio não terá estado persistente. Nesse caso, múltiplas instâncias paralelas do estágio poderá ser criada pelo ambiente de execução.

Caso contrário, a instância terá um estado persistente, porém todos os eventos para esse estágio serão consumidos de forma serial.

‘autostart’ tipo: **tabela** (atributo opcional): Se o valor desse atributo for definido, o estágio será inicializado automaticamente pelo ambiente de execução. Nesse caso, o conteúdo da tabela atribuída a esse valor será automaticamente colocada na fila de eventos do estágio antes da execução da aplicação que contém esse estágio.

‘name’ tipo: **string** (atributo opcional): Nome opcional a ser usado pelo método `tostring` do estágio.

Métodos dos objetos de estágios

`origem:connect(porta,destino,tipo)`

Equivalente a `leda.connect(origem,porta,destino,tipo)`

`estágio:push(...)`

Coloca evento na fila de eventos do estágio.

`estágio:run(...)`

Cria um grafo contendo apenas o estágio e o executa. Equivalente a `leda.graph{estágio}:run(...)`

leda.connect(origem, porta, destino, tipo)

Essa função cria um conector entre a porta de saída `porta` do estágio `origem` para a fila de eventos do estágio `destino`.

O parâmetro opcional `tipo` é uma *string* que descreve o tipo do conector a ser criado. Na versão atual, os valores possíveis são: “local” para a criação de um conector local e “decoupled” para a criação de um conector desacoplado.

Caso o conector seja local, os estágios `origem` e `destino` deverão obrigatoriamente ser mapeados para o mesmo processo. Caso o tipo do conector não seja especificado, um conector desacoplado será criado.

leda.graph({...})

Essa função cria um grafo de estágios que representa uma aplicação. Ela recebe como parâmetro um conjunto de conectores e estágios que irão compor o grafo.

Caso o parâmetro seja uma tabela com o atributo `name` definido, este será usado pelo método `tostring` do grafo.

Métodos dos objetos de grafos

`grafo:run(...)`

Inicia a execução do grafo de estágios. Recebe como parâmetro uma tabela contendo os seguintes atributos opcionais:

‘`controller`’: Um controlador a ser usado no ambiente de execução do processo atual.

‘`maxpar`’: Número máximo de instâncias de estágios definidos no grafo, caso não seja definido pelo controlador.

‘`port`’: Porta TCP a ser usada pelo processo para comunicação inter-processo.

‘`host`’: Nome do *host* da máquina que executa o processo.

`grafo:part(...)`

Cria uma partição do grafo. Essa função recebe como parâmetro uma lista ordenada de aglomerados de estágios definidos no grafo. Cada estágio deve estar definido em exatamente um aglomerado para que a partição seja válida. Essa função emite um erro de execução caso a partição não seja válida.

`grafo:map(...)`

Mapeia cada aglomerado definido anteriormente a um processo exclusivo. Cada processo remoto deve ser iniciado antes da execução da aplicação.

Processos são definidos através de uma *string* com a seguinte estrutura: “`host:porta`”.

`grafo:all()`

Retorna um aglomerado contendo todos os estágios definidos no grafo.

leda.cluster(...)

Essa função cria um aglomerado de estágios. Ela recebe como parâmetro um conjunto de estágios que irão compor o aglomerado.

Aglomerados podem ser manipulados através dos operadores aritméticos + (união), - (diferença) e * (interseção).

leda.start(...)

Inicia um novo processo e aguarda pela execução de um grafo que define um mapeamento de aglomerado para este processo.

Recebe como parâmetro uma tabela contendo os seguintes atributos opcionais:

‘`controller`’: Um controlador a ser usado no ambiente de execução do processo atual.

‘maxpar’: Número máximo de instâncias de estágios definidos no grafo, caso não seja definido pelo controlador.

‘port’: Porta TCP a ser usada pelo processo para comunicação inter-processo.

‘host’: Nome do *host* da máquina que executa o processo.

A.2

A API de Instâncias de Estágios

As funções a seguir estão disponíveis no ambiente de execução de instâncias de estágios e podem ser chamadas pelos de tratadores de eventos definidos.

leda.send(porta,...)

Encamina um evento para uma porta de saída de um estágio. Essa função retorna um valor booleano indicando o sucesso do enfileiramento do evento na fila de eventos do estágio de destino do conector correspondente.

Em caso de falha, uma *string* com o motivo da falha também é retornada.

leda.sleep(tempo)

Faz com que a instância em execução atual durma por um período de tempo (em segundos).

A *thread* é cooperativamente liberada de volta ao *pool* do processo durante esse período.

leda.quit(...)

Interrompe a execução do processo atual.

Essa função faz com que a função `leda.run` ou `leda.start` retorne os valores passados como parâmetros.

leda.gettime()

Retorna o tempo atual desde a época (segundos desde 01 de janeiro de 1970), com resolução de microsegundos.

leda.nice()

Libera cooperativamente a *thread* de volta ao *pool* do processo e coloca a instância atual de volta na fila de prontos do processo. É equivalente a `leda.sleep(0)`.

B Códigos Fonte

B.1 Teste de páginas dinâmicas do servidor HTTP

Listagem B.1: *Script* usado nos testes de páginas dinâmicas dos servidores HTTP.

```

1 local table=require "table"
2
3 local ret={
4 [[<html>
5 <head>
6   <title>CGILua loop test</title>
7 </head>
8 <body>
9 <p>Executes a long loop , used by concurrency tests</p>]]}
10
11 for i = 1, 30000 do
12   table.insert(ret ,( i.. '<br/>\n' ))
13 end
14
15 table.insert (ret , [[
16 </body>
17 </html >]])
18
19 return table.concat (ret)

```

B.2 Ordenação Distribuída de Vetores

Apresentamos aqui o código completo do algoritmo de ordenação (implementado em C) e das versões MPI e Leda da aplicação de ordenação distribuída de vetores.

Listagem B.2: Código C do algoritmo *insertion sort*.

```

1 #include <lua.h>
2 void selection_sort(double * vetor , int tam) {
3   int i,j;

```

```

4
5   for (j=0;j<tam-1;j++) {
6       int iMin=j;
7       for (i=j+1; i<tam; i++) {
8           if(vetor[i] < vetor[iMin]) iMin=i;
9       }
10      if (iMin != j) {
11          double tmp=vetor[j];
12          vetor[j]=vetor[iMin];
13          vetor[iMin]=tmp;
14      }
15  }
16 }
17
18 static int lua_operation(lua_State * L) {
19     lua_settop(L,1);
20     lua_getfield(L,1, "ptr");
21     lua_pushvalue(L,1);
22     lua_call(L,1,1);
23     lua_getfield(L,1, "length");
24     lua_pushvalue(L,1);
25     lua_call(L,1,1);
26     double * vetor=(double *)lua_touserdata(L,2);
27     int tam=lua_tointeger(L,3);
28     lua_pop(L,2);
29     selection_sort(vetor, tam);
30     return 1;
31 }
32
33 int luaopen_op(lua_State *L) {
34     lua_pushcfunção(L, lua_operation);
35     lua_pushvalue(L, -1);
36     lua_setglobal(L, "selection_sort");
37     return 1;
38 }

```

Listagem B.3: Implementação da versão MPI do algoritmo de ordenação de vetores.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include "mpi.h"
4 #define MAXPROC 120
5 #define LENGTH 1536000
6
7 #include "op.h"
8 #include <sys/time.h>

```

```

9
10 int main(int argc, char* argv[]) {
11     int i, j, np, me;
12     const int datatag = 43;
13     const int root = 0;
14     MPI_Status status;
15
16     double x[LENGTH]; /*buffer de envio*/
17     double y[LENGTH]; /*buffer de recepção*/
18
19     MPI_Init(&argc, &argv);
20     MPI_Comm_size(MPLCOMM_WORLD, &np);
21     MPI_Comm_rank(MPLCOMM_WORLD, &me);
22
23     if (np>MAXPROC || np%2 != 0) {
24         if (me == 0) {
25             printf("Voce deve usar um numero par de processos (<=
                %d)\n", MAXPROC);
26         }
27         MPI_Finalize();
28         exit(0);
29     }
30
31     struct timeval tv={0,0};
32     gettimeofday(&tv, NULL);
33     double init_time=tv.tv_sec+(((double)tv.tv_usec)/1000000.0);
34
35     if (me == 0) {
36         for (i=0; i<LENGTH; i++) {
37             x[i] = i;
38         }
39
40         MPI_Scatter(x, LENGTH/np, MPLDOUBLE, y, LENGTH/np,
                    MPLDOUBLE, root, MPLCOMM_WORLD);
41
42         for (i=1; i<np; i++) {
43             MPI_Recv (y, LENGTH/np, MPLDOUBLE, i, datatag,
                    MPLCOMM_WORLD, &status);
44         }
45
46         gettimeofday(&tv, NULL);
47         double end_time=tv.tv_sec+(((double)tv.tv_usec)/1000000.0);
48         printf("mpi\t%d\t%d\t%d\t%.6lf\n",LENGTH,np,LENGTH/np,end_time-init_time);
49     } else {
50         MPI_Scatter(x, LENGTH/np, MPLDOUBLE, y, LENGTH/np,
                    MPLDOUBLE, root, \
51         MPLCOMM_WORLD);

```

```

52
53     selection_sort(y,LENGTH/np);
54
55     MPI_Send (y, LENGTH/np, MPI.DOUBLE, 0, datatag,
56             MPI.COMM_WORLD);
57 }
58 MPI_Finalize();
59 exit(0);
60 }

```

Listagem B.4: Implementação da versão Leda do algoritmo de ordenação de vetores.

```

1  require 'leda'
2  local np=tonumber(assert(arg[1]))
3  local length=arg[2] or 1536000
4  local procs=require 'hosts'
5
6  local s=leda.stage "scatter" {
7    init=function() memarray=require 'leda.memarray' end,
8    handler = function()
9      local tempo_ini=leda.gettime()
10     local acc=length
11     for proc=1,np do
12       local vetor=memarray(length/np)
13       for i=1,(length/np) do
14         vetor[i]=acc acc=acc-1
15       end
16       leda.send(proc,vetor,tempo_ini)
17     end
18   end,
19   autostart=true
20 }
21
22 local op=leda.stage "Executa operacao" {
23   init = function() selection_sort=require 'op' end,
24   handler = function(vetor,...)
25     selection_sort(vetor)
26     leda.send(1,vetor,...)
27   end
28 }
29
30 local reduce=leda.stage "reducer" {
31   init=function() i=0 res={} end,
32   handler = function(vetor,tempo_ini)
33     i=i+1

```

```
34     res [i]=vetor
35     if np == i then
36         print ("leda ",length ,np,vetor:length(),leda.gettime()-tempo_ini)
37         leda.quit()
38     end
39 end,
40     serial=true
41 }
42
43 local g=leda.graph{}
44 local clusters={leda.cluster(s,reduce)}
45 for i=1,np do
46     local opn=leda.stage(op.name..i)(op)
47     g:add(s:connect(i,opn))
48     g:add(opn:connect(1,reduce))
49     table.insert(clusters,leda.cluster(opn))
50 end
51 g:part(unpack(clusters)):map(unpack(procs))
52 g:run()
```
