SUPORTE A RECONFIGURAÇÃO DINÂMICA EM APLICAÇÕES DE PROCESSAMENTO DISTRIBUÍDO DE FLUXOS DE DADOS

RODRIGO SILVA OLIVEIRA

SUPORTE A RECONFIGURAÇÃO DINÂMICA EM APLICAÇÕES DE PROCESSAMENTO DISTRIBUÍDO DE FLUXOS DE DADOS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: RENATO ANTÔNIO CELSO FERREIRA

Belo Horizonte

05 de setembro de 2012

Oliveira, Rodrigo Silva.
O48s Suporte a reconfiguração dinâmica em aplicações de processamento distribuído de fluxos de dados / Rodrigo Silva Oliveira — Belo Horizonte, 2012.
xxiv, 88 f.: il.; 29cm.

Dissertação (mestrado) — Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Renato Antônio Celso Ferreira.

1. Computação – Teses. 2. Sistemas operacionais distribuídos (Computadores) – Teses. 3. Computação de alto desempenho – Teses. I. Orientador. II. Título.

519.6*22(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Suporte a reconfiguração dinâmica em aplicações de processamento distribuído de fluxos de dados

RODRIGO SILVA OLIVEIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. RENATO ANTÔNIO CELSO FERREIRA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. DORGIVAL OLAVO GUEDES NETO
Departamento de Ciência da Computação - UFMG

DR. LUIZ EDUARDO DA SILVA RAMOS
Bolsista Pós-Doc/DCC - UFMG

PROF. WAGNER MEIRA JÚNIOR
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 05 de setembro de 2012.

Dedico este trabalho aos meus pais, Joaquim e Geralda, ao meu grande amor, Thatyene, ao meu irmão, Rangel, aos professores, aos colegas de curso e a todos aqueles que acreditam na educação como meio de transformação social.

Agradecimentos

Agradeço a Deus pela vida, pela família e amigos que tenho e por todas as graças recebidas.

Aos meus pais pelo amor incondicional, pelo exemplo de vida e pela confiança.

À minha noiva, Thatyene, pelo apoio, pelo carinho e por ter feito parte deste trabalho, ajudando-me e motivando-me quando tudo parecia dar errado.

Ao meu irmão, Rangel, que esteve ao meu lado nos momentos mais difíceis desse mestrado, apoiando-me e ajudando-me a solucionar problemas muito complexos encontrados na implementação do ambiente Watershed.

Aos professores Renato Antônio Celso Ferreira e Wagner Meira Jr. pelas orientações, oportunidades, conselhos e provocações construtivas.

Ao professor Arnaldo de Albuquerque Araújo, por ter acreditado em meu trabalho e pelas grandes oportunidades que que me ofereceu.

Aos demais professores do Departamento de Ciência da Computação da UFMG por minha formação e pela disponibilidade.

A todos os funcionários do DCC e do ICEx pela eficiência e gentileza no relacionamento com os alunos.

Ao meu amigo Luiz Eduardo da Silva Ramos, que empenhou-se na leitura desse texto, dando contribuições valiosas na fase conclusiva do trabalho.

À minha grande amiga Ana Paula Brandão Lopes, que me ensinou muito sobre pesquisa quando trabalhamos juntos.

Ao meu amigo Fillipe Dias Moreira de Souza, que me incentivou e me mostrou que eu era capaz de alcançar qualquer objetivo, independente dos obstáculos a serem contornados.

À "Dra." Caroline, por ser um excelente exemplo do tipo de pessoa que eu não quero e não devo ser e por estar completamente errada sobre a diferença entre Brasil e Dinamarca.

Aos colegas da graduação e mestrado pela convivência e troca de experiências.

Aos meus amigos do laboratório NPDI, que sempre respeitaram meu trabalho e

que sempre mostraram-se solidários quando necessário.

Aos meus amigos do laboratório e-SPEED pelas dicas, pelo companheirismo e pelos ótimos momentos que passamos juntos.

À equipe de desenvolvimento da biblioteca Open MPI pela ajuda prestada.

E finalmente, agradeço às entidades CAPES, CNPq, InWeb, FAPEMIG e SERPRO pelo incentivo à pesquisa e pelo auxílio financeiro que possibilitou a realização deste trabalho.

"Inexiste no mundo coisa mais bem distribuída que o bom senso, visto que cada indivíduo acredita ser tão bem provido dele que mesmo os mais difíceis de satisfazer em qualquer outro aspecto não costumam desejar possuí-lo mais do que já possuem. E é improvável que todos se enganem a esse respeito; mas isso é antes uma prova de que o poder de julgar de forma correta e discernir entre o verdadeiro e o falso, que é justamente o que é denominado bom senso ou razão, é igual em todos os homens; e, assim sendo, de que a diversidade de nossas opiniões não se origina do fato de serem alguns mais racionais que outros, mas apenas de dirigirmos nossos pensamentos por caminhos diferentes e não considerarmos as mesmas coisas. Pois é insuficiente ter o espírito bom, o mais importante é aplicá-lo bem. As maiores almas são capazes dos maiores vícios, como também das maiores virtudes, e os que só andam muito devagar podem avançar bem mais, se continuarem sempre pelo caminho reto, do que aqueles que correm e dele se afastam. "

(René Descartes - Discurso sobre o Método)

Resumo

Os avanços científicos e tecnológicos observados nas últimas décadas têm como um de seus resultados a geração de coleções de dados cada vez maiores. Em alguns cenários, essas coleções são representadas na forma de fluxos, que podem ser definidos como sequências de dados com ordenação temporal. Exemplos incluem estatísticas de tráfego de rede, medições de monitoração de ambientes por sensores, dados coletados por satélites e, em ascendência acelerada, dados disponibilizados em redes sociais *online* e serviços da *Web*. Uma vez que a análise de dados figura como etapa fundamental para a compreensão e solução de diversos problemas, ferramentas apropriadas e eficientes para o processamento de tais coleções fazem-se necessárias.

Nesse contexto, a utilização de sistemas distribuídos tem se popularizado, constituindo uma solução robusta e eficiente para a construção de aplicações de processamento de fluxos de dados. Essas aplicações geralmente executam por longos períodos em ambientes compartilhados ou dinâmicos, estando sujeitas a uma série de mudanças que podem ocorrer ao longo do tempo, tais como falhas, aumento/diminuição de recursos computacionais e variação na carga de trabalho. A natureza dinâmica dessas aplicações demanda a existência de mecanismos que permitam adaptações em tempo de execução, como forma de acomodar as mudanças ocorridas no ambiente. Neste trabalho, é proposto um mecanismo de assistência a reconfigurações dinâmicas para aplicações de processamento distribuído de fluxos de dados. O mecanismo foi implementado como uma funcionalidade de Watershed, que é um sistema de processamento distribuído para aplicações construídas no modelo de programação filtro-fluxo. Os resultados obtidos na avaliação experimental mostram que o mecanismo possibilita, de forma eficiente e consistente, a execução das operações de reconfiguração propostas.

Palavras-chave: Sistemas Distribuídos, Computação de Alto Desempenho, Processamento de Fluxos de Dados, Reconfiguração Dinâmica.

Abstract

The scientific and technological advances observed in last decades have as a result the generation of increasing data collections. In some scenarios, these collections are represented as streams, which can be defined as sequences of data in time ordering. Examples include network traffic statistics, sensor measurements used for environments monitoring, data collected by satellites, and in accelerated growth, data available on online social networks and services on the Web. Once data analysis is a fundamental step to understanding and solving various problems, appropriate and efficient tools for processing such collections are necessary.

In this context, the use of distributed systems has become popular, providing a robust and efficient solution to build stream processing applications. These applications often run for long time in shared or dynamic environments, being subject to a series of changes that may occur over time, such as faults, increase/decrease of computational resources and variation in workload. The dynamic nature of these applications requires the existence of mechanisms that allow adaptations at runtime, in order to accommodate the changes in the environment. This work proposes a mechanism to support dynamic reconfiguration for distributed data stream processing applications. The mechanism is implemented as a functionality of Watershed, which is a distributed processing system for applications built in the filter-stream programming model. The results of the experimental evaluation show that the mechanism efficiently and consistently supports the proposed reconfiguration operations.

Keywords: Distributed Systems, High Performance Computing, Data Stream Processing, Dynamic Reconfiguration.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

API *Application Programming Interface* 23

DSM *Distributed Shared Memory*

DTD *Document Type Definition* 25

E/S Entrada e Saída 7

MPI *Message Passing Interface* 15

NFS *Network File System* 62

PVM *Parallel Virtual Machine* 9

TCP *Transmission Control Protocol* 11

XML *Extensible Markup Language* 9

Sumário

Capítulo 1

Introdução

A análise de dados é um processo fundamental para a compreensão e solução de muitos problemas teóricos e práticos. Esse processo envolve, em geral, as etapas de limpeza, transformação e modelagem de dados, com o objetivo de evidenciar informações úteis para a elaboração de conclusões e para a tomada de decisões, com as respectivas ações de resposta. O acelerado progresso científico e tecnológico observado nas últimas décadas tem como um de seus resultados a geração de coleções de dados cada vez maiores. O processamento dessas coleções de dados por um único computador é, na maioria das vezes, inviável, haja vista os requisitos de desempenho estabelecidos pelas aplicações das mais variadas áreas do conhecimento.

Diante de tais requisitos, a utilização de arquiteturas cooperativas envolvendo um grande volume de recursos computacionais é reconhecida como sendo uma estratégia eficaz e eficiente. Alguns fatores que impulsionam a utilização de tais arquiteturas são o baixo custo dos componentes de processamento, a evolução das tecnologias das redes de computadores e a popularização dos processadores *multi-core*. Tais fatores possibilitam o processamento massivamente paralelo com eficiência na comunicação de dados entre os recursos computacionais. Nesse contexto, é comum que se tenha aplicações distribuídas executando em *clusters* com centenas e até milhares de servidores de processamento de propósito geral. Como resultado, arquiteturas de processamento paralelo e distribuído têm se mostrado muito interessantes no contexto de processamento de grandes volumes de dados com requisitos de alto desempenho.

Recentemente, fontes de dados *online* têm tomado forma de fluxos de dados, ou seja, uma lista infinita de elementos provenientes de um conjunto de dados de interesse com o conceito de ordenação temporal [Stephens, 1997; Gedik et al., 2008]. Exemplos de fluxos de dados incluem estatísticas de tráfego de rede em sistemas de comunicação, medidas geradas pela monitoração de ambientes por sensores, dados coletados por

satélites e, em ascendência acelerada, dados disponibilizados em redes sociais e serviços da *Web*.

A proliferação dessas fontes de dados tem motivado uma mudança na forma de processamento. Tal mudança está basicamente fundamentada no fato de que a estratégia de armazenamento com processamento posterior, utilizada nos sistemas tradicionais de bancos de dados, não é a mais apropriada para o novo cenário, uma vez que há, em geral, o requisito de que o processamento seja realizado à mesma taxa com que os fluxos de dados chegam ao sistema. Um modelo apropriado deve ter como premissa o processamento sob demanda, onde os dados presentes em um fluxo são processados à medida em que chegam ao sistema, sendo que o resultado de tal processamento pode ser também um fluxo de dados.

Sistemas de processamento de fluxos de dados [Stephens, 1997] são sistemas compostos por um conjunto de módulos que processam em paralelo e que se comunicam por meio da transferência de dados através dos canais existentes entre eles. Esses sistemas podem ser modelados como grafos direcionados, contendo ou não ciclos, como mostra a Figura 1.1.

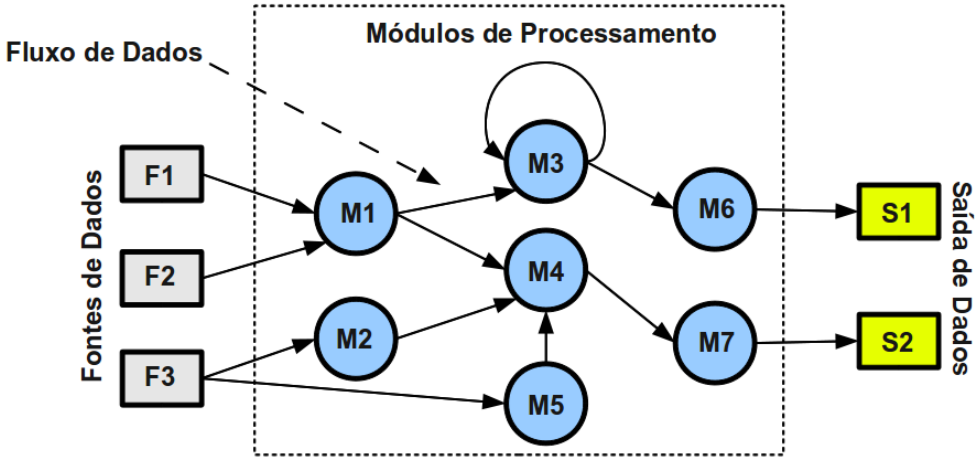


Figura 1.1: Arquitetura típica de um sistema de processamento de fluxos de dados.

Na figura podemos identificar três categorias básicas de componentes:

- *Fontes de dados.* Componentes responsáveis pela captura dos dados do ambiente externo ao sistema de processamento. Realizam também a formatação dos dados para que os mesmos possam ser processados.
- *Módulos de processamento.* Componentes responsáveis por processar os dados de acordo com a lógica da aplicação. Podem ser organizados em diversos estágios, contendo ou não ciclos.

- *Saída de dados.* Componentes cuja função é formatar os resultados finais do processamento, de forma que possam ser entregues ao usuário final.

A motivação usual para tais sistemas é alcançar o máximo de desempenho, oferecendo soluções escaláveis para o processamento dos dados, considerando os seguintes aspectos:

- *Paralelismo.* Melhorar o desempenho das aplicações, explorando ao máximo o potencial de paralelismo presente nas mesmas.
- *Uso de Recursos.* Fazer uso eficiente dos recursos disponíveis, tirando o máximo proveito do poder computacional agregado.
- *Interface de Programação.* Prover uma abstração que torne fácil a tarefa do programador, isentando-o das responsabilidades inerentes ao gerenciamento das primitivas de processamento paralelo e distribuído.

As aplicações de processamento distribuído de fluxos de dados geralmente executam por longos períodos e, na maioria das vezes, utilizam ambientes compartilhados ou dinâmicos. Por tal motivo, essas aplicações estão sujeitas a uma série de mudanças que podem ocorrer no ambiente. Tais mudanças podem ser ocasionadas por situações de falhas, disponibilização de novos recursos de processamento e alterações na carga de trabalho dos recursos existentes. Essa característica faz com que seja necessário algum mecanismo de apoio a reconfigurações dinâmicas. Uma reconfiguração dinâmica pode ser entendida como uma mudança na topologia ou na granularidade de paralelismo da aplicação como forma de resposta às mudanças ocorridas no ambiente de execução.

Várias estratégias de reconfiguração dinâmica têm sido propostas, sendo que todas compartilham objetivos e ações semelhantes [Stellner, 1996; Bhandarkar et al., 2001; El Maghraoui et al., 2006]. Com relação aos objetivos, tais mecanismos visam aumentar o desempenho das aplicações, utilizando de forma mais eficiente os recursos computacionais. Esse objetivo é alcançado por meio da expansão ou retração de aplicações, o que é feito com o aumento ou redução do número de elementos de processamento, respectivamente. Outra maneira de se obter melhoria no desempenho de aplicações distribuídas é utilizar estratégias de migração de processos de máquinas mais sobrecarregadas para máquinas mais ociosas, de forma que a carga de trabalho seja melhor balanceada. Outro objetivo em comum é minimizar o impacto desses mecanismos no desempenho geral das aplicações executando em ambientes distribuídos.

Muitas dessas abordagens carecem de abstrações que tornem o trabalho do programador de aplicação menos complexo. Algumas das tarefas delegadas ao programa-

dor de aplicação em situações de reconfiguração incluem a redistribuição de estados de processos e a reorganização da estratégia de troca de mensagens. Essas tarefas não fazem parte do domínio da aplicação e, como tal, não devem ser realizadas pelo programador de aplicação. Dessa forma, a estratégia mais interessante é aquela na qual o sistema de processamento as realiza de maneira transparente.

Em resposta às demandas mencionadas, nosso objetivo nesse trabalho é propor um mecanismo transparente que possibilita a reconfiguração dinâmica de aplicações de processamento distribuído de fluxos de dados. Esse mecanismo é parte integrante do ambiente Watershed, que é resultado de trabalhos recentes do grupo de pesquisa em sistemas distribuídos do DCC/UFMG, sendo o autor dessa dissertação corresponsável pelo seu projeto e desenvolvimento.

1.1 Definição do Problema

A natureza dinâmica das aplicações de processamento distribuído de fluxos de dados e dos ambientes de computação compartilhados são fatores preponderantes no que se refere a desempenho e escalabilidade. Em particular, é comum ocorrerem mudanças significativas ao longo do tempo, tanto na infraestrutura disponível, quanto nas demandas por processamento. Exemplos dessas mudanças incluem falhas de máquinas, adição de serviços, remoção de serviços, alteração no volume de dados a serem processados, entre muitas outras.

Nesse contexto, é essencial que a estrutura de uma aplicação possa ser modificada em tempo de execução em resposta às mudanças experimentadas, como tentativa de manter o nível de desempenho. Um mecanismo de reconfiguração dinâmica é um mecanismo que possibilita as modificações estruturais necessárias. Tal mecanismo deve ser capaz de efetivar ações como adição, remoção e migração de processos, ao passo que deve manter a semântica e a correção da aplicação, impactar pouco no desempenho do ambiente como um todo e ser o mais transparente possível ao programador de aplicação.

O problema abordado no presente trabalho pode ser enunciado, resumidamente, da seguinte maneira:

> *Dada uma aplicação de processamento distribuído de fluxos de dados, promover, em tempo de execução, de forma eficiente e consistente, as funcionalidades de adição, remoção e migração dos processos que a compõem.*

É importante ressaltar que o escopo desse trabalho é limitado ao mecanismo de execução das reconfigurações dinâmicas. Portanto, não serão abordadas questões

relacionadas à monitoração e tomada de decisões sobre a necessidade de tais reconfigurações.

1.2 Contribuições

A principal contribuição desse trabalho é a descrição do projeto e implementação de um mecanismo simples, eficaz e eficiente para o apoio a reconfigurações dinâmicas de aplicações de processamento distribuído de fluxos de dados. O mecanismo proposto integra técnicas presentes na literatura, adaptadas e aplicadas ao contexto de sistemas de computação em fluxos de dados. Dessa forma, o trabalho contribui para o aprimoramento do estado da arte na área, uma vez que apresenta soluções inovadoras ao problema formulado.

Outra contribuição importante é a participação na proposta do ambiente Watershed, que é um ambiente de processamento distribuído, equipado com mecanismos dinâmicos para o gerenciamento de aplicações em ambientes compartilhados.

Por fim, o trabalho apresenta uma avaliação experimental detalhada, demonstrando a efetividade e a eficiência do mecanismo proposto. Os resultados obtidos reforçam a hipótese de que é viável modificar, em tempo de execução, a estrutura de aplicações distribuídas concorrentes, sendo possível dimensionar o uso de recursos no sistema sem prejuízo significativo ao seu desempenho global.

1.3 Organização deste Documento

O restante deste documento é organizado da seguinte maneira. O capítulo 2 apresenta uma revisão bibliográfica sobre computação em fluxos de dados e reconfiguração dinâmica de aplicações distribuídas. Em seguida, no capítulo 3, é feita a descrição do ambiente de processamento distribuído Watershed, abordando sua arquitetura, modelo de programação, operação e resultados experimentais. A proposta do mecanismo de reconfiguração dinâmica de aplicações Watershed é apresentada no capítulo 4. O capítulo 5 detalha a avaliação experimental do mecanismo proposto e discute os resultados obtidos. As conclusões gerais e tópicos de trabalhos futuros são discutidos no capítulo 6.

Capítulo 2

Trabalhos Relacionados

Esse capítulo apresenta uma revisão da literatura sobre o tema abordado na dissertação. Os trabalhos relacionados foram divididos em duas partes: a seção 2.1 descreve um conjunto de trabalhos significativos sobre sistemas de computação em fluxos de dados e a seção 2.2 apresenta as técnicas mais utilizadas na reconfiguração dinâmica de aplicações distribuídas.

2.1 Computação em Fluxos de Dados

Computação em fluxos de dados é uma expressão amplamente utilizada na literatura para descrever uma variedade de sistemas de computação paralela. Todos esses sistemas compartilham uma característica bem definida, que é o fato de a computação ser decomposta em uma coleção de módulos que processam em paralelo e que trocam dados de forma contínua através de canais de comunicação [Burge, 1975; Stephens, 1997; Tanenbaum & Steen, 2006].

O termo "fluxo" (do inglês, *stream*) foi utilizado pela primeira vez em Ciência da Computação na década de 1960 [Landin, 1965a,b] para modelar o processamento de listas na linguagem ALGOL 60. Naquele mesmo trabalho, o autor menciona a possibilidade de utilização de um modelo semelhante para o sistema de E/S (Entrada e Saída) da linguagem, o que pode ser considerado o primeiro indício de fluxos de dados tal como é conhecido atualmente.

A partir da década de 1960, quando os primeiros trabalhos de pesquisa em computação de fluxos de dados surgiram, a área tornou-se extremamente ativa e desafiadora. O avanço tecnológico trouxe novas arquiteturas de computadores e tarefas cada vez maiores, fazendo com que a demanda por sistemas de computação paralela fosse intensificada. É nesse contexto que os modelos de computação em fluxos de dados ganham

evidência, mostrando-se apropriados em diversos cenários de pesquisa e indústria. A seguir serão apresentados os trabalhos de maior evidência no contexto de computação distribuída em fluxos de dados.

2.1.1 DataCutter

DataCutter [Beynon et al., 2001, 2002; Spencer et al., 2002] é um arcabouço projetado para permitir a análise de coleções de dados científicos em ambientes distribuídos e heterogêneos. O modelo de programação adotado por esse sistema, denominado filtro-fluxo (do inglês, *filter-stream*), representa os estágios de uma aplicação intensiva em dados como um conjunto de filtros. Um filtro é um objeto de processamento, definido pelo usuário, que realiza alguma transformação nos dados da aplicação. Cada filtro pode ter várias réplicas que podem ser executadas em diferentes computadores dentro do ambiente de rede, sendo que a troca de informações entre esses filtros dá-se por meio de fluxos de dados unidirecionais.

Um aspecto importante desse modelo de programação é a transparência oferecida pelo ambiente de execução, uma vez que a alocação de recursos, a replicação de filtros e a manutenção dos canais de comunicação são feitas de forma automatizada. O alto desempenho observado em aplicações DataCutter é resultado direto dos dois níveis de paralelismo oferecidos pelo modelo de programação. O primeiro deles diz respeito ao paralelismo de tarefas, que surge com a decomposição das aplicações em um conjunto de filtros que representam os estágios de processamento. O outro nível de paralelismo refere-se ao paralelismo de dados, que tem origem na criação de várias réplicas para cada filtro.

A topologia de uma aplicação DataCutter deve ser definida de forma estática por meio de arquivos de configuração. Em outras palavras, todas as conexões entre filtros devem ser descritas antes da carga da aplicação, não podendo ser alteradas em tempo de execução. Outra limitação do sistema é a impossibilidade de execução simultânea de diferentes aplicações. Essas limitações tornam difícil a utilização de DataCutter em ambientes dinâmicos, com compartilhamento de resultados por aplicações diferentes.

2.1.2 Anthill

Outro ambiente de processamento de fluxos de dados baseado no modelo filtro-fluxo é o Anthill [Ferreira et al., 2005], que encapsula um conjunto de abstrações que permitem aos programadores construírem aplicações distribuídas de forma fácil e intuitiva. As abstrações oferecidas permitem explorar tanto o paralelismo de tarefas quanto

o paralelismo de dados, utilizando a decomposição das aplicações em estágios (filtros) e a replicação desses estágios em instâncias idênticas. Os canais de comunicação de dados entre filtros são estabelecidos no momento de carga da aplicação, sendo a transferência de dados realizada utilizando-se o PVM (*Parallel Virtual Machine*) [Sunderam, 1990] como infraestrutura básica de comunicação. As políticas de distribuição de mensagens entre instâncias de filtros existentes são: (i) *broadcast*, onde as mensagens são enviadas para todas as cópias de um filtro consumidor, (ii) *round-robin*, onde as mensagens são enviadas seguindo um esquema de revezamento entre as cópias de um filtro e (iii) *labeled stream*, onde as mensagens são direcionadas a cópias específicas de filtros segundo uma função de mapeamento aplicada sobre os dados da mensagem.

Anthill possibilita a execução de uma aplicação por vez, sem compartilhamento de resultados ou filtros de execução. A topologia de uma aplicação é definida de maneira estática de acordo com um arquivo XML (*Extensible Markup Language*) de descrição, que contém a estrutura do grafo de processos e as políticas de distribuição de mensagens a serem utilizadas em cada canal de comunicação.

Algumas funcionalidades foram adicionadas à plataforma básica de Anthill tornando-o um ambiente mais robusto e eficiente. Dentre essas inovações, merecem destaque:

- *Reconfiguração* [Fireman et al., 2008]. Mecanismo que permite a adição de instâncias de filtros em tempo de execução. Utiliza uma estratégia de manutenção de estado global distribuído, implementado por meio de um sistema de memória compartilhada distribuída (do inglês, *Distributed Shared Memory* – DSM) para permitir o acesso a dados globais por instâncias distintas de maneira consistente.

- *Orientação a eventos* [Teodoro et al., 2008]. Mecanismo de enfileiramento e controle sobre as mensagens de entrada dos filtros em execução. Esse mecanismo apresenta uma série de benefícios relacionados à possibilidade de exploração de paralelismo intra-máquina, utilizando o poder de processamento das arquiteturas *multi-core* para o processamento paralelo de mensagens. Exemplos dessas possibilidades incluem o processamento de mensagens de um mesmo fluxo utilizando memória compartilhada e o processamento paralelo de mensagens provenientes de fluxos de dados distintos, o que é feito por meio de funções *callback* definidas pelo programador.

- *Computação em ambientes heterogêneos* [Teodoro et al., 2010]. Estratégias de execução eficiente de aplicações em ambientes heterogêneos equipados com GPUs e CPUs *multi-core*. O trabalho faz uma análise em tempo de execução das tarefas

ativas no ambiente, sendo capaz de prever, para cada uma ou parte da mesma, qual dispositivo é mais apropriado para sua execução.

2.1.3 MapReduce

MapReduce [Dean & Ghemawat, 2004] é um modelo de programação desenvolvido pela Google para o processamento e geração de grandes bases de dados. Esse modelo foi implementado e é base para o funcionamento de muitas aplicações mantidas pela empresa. Nesse modelo, os usuários devem especificar funções de mapeamento (do inglês, *map*) e funções de redução (do inglês, *reduce*). As funções de mapeamento devem processar pares no formato `(chave, valor)` e produzir como resultado um conjunto intermediário de resultados, também organizados em pares `(chave, valor)`. Funções de redução devem ser implementadas de maneira que façam a junção de todos os valores indexados pela mesma chave no conjunto intermediário, gerando como resultado final pares contendo uma chave e o resultado da função de redução aplicada sobre a lista de valores intermediários. Esse processo é ilustrado na figura 2.1.

MAPEAMENTO	$(k1,\ v1)$	$\Longrightarrow$	$lista\ (k2,\ v2)$
REDUÇÃO	$(k2,\ lista\ (v2))$	$\Longrightarrow$	$(k3, f(lista\ (v2)))$

Figura 2.1: Decomposição de aplicações nas etapas de mapeamento e redução no modelo de computação MapReduce.

A importância desse modelo é que muitas aplicações reais podem ser expressadas segundo funções de mapeamento e redução, potencializando sua utilização em diferentes áreas, conforme observam os próprios autores.

Aplicações MapReduce podem ser automaticamente paralelizadas e executadas em grandes *clusters* de forma distribuída. Assim como DataCutter, o ambiente de execução cuida dos detalhes de particionamento de dados, escalonamento de processos, tolerância a falhas e comunicação entre máquinas. Um detalhe importante na implementação da Google é que tanto a aquisição dos dados de entrada pelas funções de mapeamento quanto a transferência de dados entre as funções é feita por meio do sistema de arquivos Google *File System* [Ghemawat et al., 2003]. Essa estratégia se mostra interessante para o caso do MapReduce pelo fato de que as operações são geralmente simples, mas aplicadas sobre grandes volumes de dados, sem que haja troca intensiva de mensagens. Isso nem sempre é verdade no modelo filtro-fluxo onde há um número expressivo de troca de pequenas mensagens entre as unidades de processamento. Outra diferença significativa entre o modelo filtro-fluxo e o modelo empregado

em MapReduce é que nesse último as aplicações podem ser decomposta em apenas dois estágios de processamento, o que não ocorre no modelo filtro-fluxo, que não define limite para o número de estágios.

2.1.4 Dryad

Isard et al. [2007] propuseram Dryad, um ambiente de execução distribuída de propósito geral para aplicações com paralelismo de dados de grão grosso e grão fino. Uma aplicação escrita para tal ambiente pode ser vista como um grafo acíclico de fluxos de dados, combinando vértices de processamento com arestas de comunicação. A comunicação entre os vértices de processamento é feita de formas diferentes, como arquivos (sistema de armazenamento distribuído), canais TCP (*Transmission Control Protocol*) e memória compartilhada, dependendo da adequação ao tipo de comunicação demandado. O sistema faz o escalonamento dos vértices de processamento para serem executados em múltiplos computadores ou nas unidades de processamento de CPUs *multicore*. Além da decomposição das aplicações em estágios de processamento, são criadas múltiplas cópias de cada vértice de processamento, explorando o paralelismo de dados. Ao contrário do modelo utilizado pelo MapReduce, Dryad permite a criação de quantos estágios de processamento forem necessários para a aplicação. Uma limitação clara desse modelo é que as aplicações não podem ser representadas como grafos cíclicos.

A figura 2.2 mostra a arquitetura do sistema Dryad. Uma tarefa Dryad é coordenada por um processo chamado "gerente de tarefa" (JM) que executa em uma das máquinas do *cluster* ou na máquina do usuário, desde que essa tenha acesso ao *cluster* via rede. O gerente é responsável unicamente por controlar a execução das tarefas, não sendo intermediador de comunicação entre os vértices. Ele contém o código da aplicação para construir o grafo de comunicação de tarefas de forma que possam ser escalonadas.

A arquitetura tem um servidor de nomes (NS) que pode ser usado para encontrar os nós disponíveis. Esse servidor também informa a localização de cada computador dentro da rede de forma que decisões de escalonamento possam levar em consideração o critério de localidade. Existe um processo *daemon* (D) executando em cada nó que é responsável por criar processos ao comando do gerente de tarefa. A primeira vez que um vértice de processamento (V) é executado em um computador, seu código binário é enviado pelo gerente de tarefa para o *daemon* escolhido. Nas demais vezes, o código fonte é recuperado de um mecanismo de *cache*. O *daemon* é utilizado também para intermediar a comunicação entre o gerente de tarefa e os vértices de processamento, monitorando a execução de cada vértice. O escalonador de tarefas executa um enfilei-

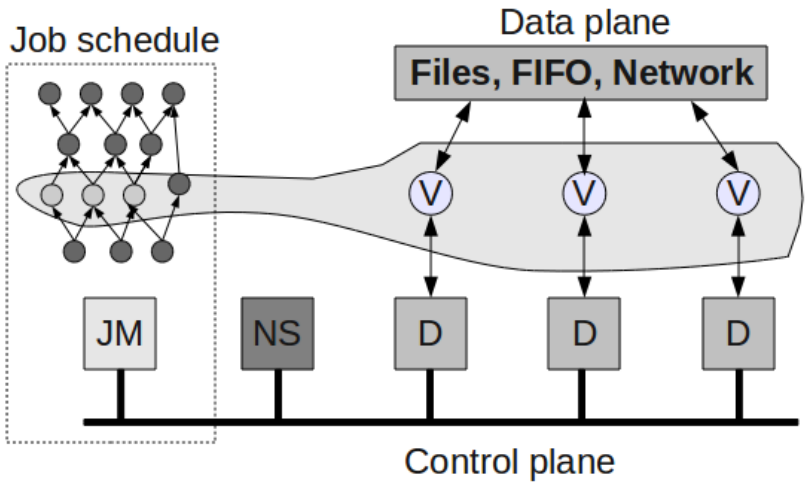


Figura 2.2: Arquitetura do sistema Dryad [Isard et al., 2007].

ramento simples das tarefas de execução em lote e utiliza uma estratégia gulosa para a escolha de nós a hospedarem vértices de execução, partindo da restrição de que apenas uma aplicação executa por vez no ambiente.

2.1.5 StreamIt

StreamIt [Thies et al., 2001, 2002, 2005] consiste em uma linguagem e um compilador projetados especificamente para a programação no paradigma de processamento de fluxos de dados. A linguagem de StreamIt provê abstrações de alto nível para a programação de fluxos de dados, ao passo que o compilador realiza otimizações específicas para aumentar o desempenho das aplicações. A linguagem especificada é baseada no modelo filtro-fluxo e oferece três maneiras de comunicação: (i) *Pipeline*, que permite o encadeamento de filtros; (ii) *SplitJoin*, que é usado para especificar fluxos paralelos independentes que divergem de um módulo de particionamento de itens de dados e que são agrupados em módulos de junção e (iii) *FeedbackLoop*, que possibilita a criação de ciclos no grafo de fluxos. O poder de expressão de StreamIt possibilita que vários níveis de paralelismo sejam explorados, como feito por Gordon et al. [2006], que descrevem um sistema de compilação que combina o paralelismo de dados, tarefas e *pipeline* para alcançar um melhor desempenho em arquiteturas *multi-core*.

Embora aplicações StreamIt possam ser paralelizadas automaticamente pelo compilador proposto, a linguagem é de difícil utilização para aplicações de propósito geral devido à falta de tipos de dados adequados e ao modelo de programação restrito.

2.1.6 TelegraphCQ

O projeto TelegraphCQ [Madden et al., 2002; Chandrasekaran et al., 2003] consiste em uma ferramenta de arquitetura adaptativa para o processamento de grandes volumes de dados no formato de fluxos. Os usuários têm à disposição uma linguagem semelhante a SQL para o processamento contínuo de *queries* aplicadas a janelas temporais dentro dos fluxos de dados. A linguagem possui um grande número de construções, que podem ser compostas para a construção de aplicações complexas e flexíveis.

O modelo de aplicação para essa arquitetura baseia-se no fato de que, ao contrário das aplicações tradicionais, onde pode ser assumido que os dados residem em locais conhecidos, os dados das novas aplicações encontram-se sempre em movimento e sofrem mudanças constantes. No cenário explorado por TelegaphCQ, os processadores de *queries* devem reagir à chegada dos dados, processando-os e gerando algum resultado parcial. Já em sistemas tradicionais, a chegada de uma *query* ao sistema dispara acesso aos dados armazenados para o atendimento da requisição. Essa diferença caracteriza basicamente a mudança de paradigma vivenciada nas aplicações de processamento de grandes volumes de dados modernas.

Arquitetonicamente, TelegraphCQ é constituído por três módulos principais: *Ingress and Cache* (responsável pelo tratamento da entrada de dados no sistema), *Query Processing* (aplicação de operadores relacionais aos dados) e *Adaptative Routing* (tomada de decisões sobre o roteamento de mensagens entre os elementos de processamento de *queries*). Esse último módulo é muito importante, pois contém algoritmos para a otimização da ordem de aplicação dos operadores na próxima etapa do processamento, podendo realizar trocas entre operadores comutativos, por exemplo.

Essa ferramenta é específica para o processamento utilizando operações da álgebra relacional, não oferecendo recursos para o processamento de propósito geral, como é comum em grande parte das ferramentas de processamento de fluxos de dados.

2.1.7 Infosphere

InfoSphere [Pu et al., 2001; Koster et al., 2001] é uma plataforma de computação de alto desempenho desenvolvida pela IBM com o objetivo de possibilitar a construção de aplicações estritamente orientadas a informações. A plataforma tem foco na definição de nível de qualidade de serviço para os elementos de processamento distribuídos que interagem por meio de fluxos de dados, o que é uma característica interessante e inovadora.

A plataforma permite que aplicações de usuários consumam, analisem e correlacionem informações provenientes de centenas de fontes de dados, possivelmente hetero-

gêneas, em tempo real. Essa análise é feita de forma contínua sobre volumes massivos de dados a uma taxa de *Petabytes* por dia. Adicionalmente, InfoSphere possui um mecanismo de adaptação a mudanças repentinas nos formatos e tipos de dados dos fluxos ativos, realizando os ajustes necessários para a manutenção do nível de qualidade de serviço estabelecido para as aplicações.

Do ponto de vista do programador, a plataforma permite o desenvolvimento rápido de novas aplicações por meio de um componente declarativo denominado SPADE [Amini et al., 2006; Gedik et al., 2008]. Dentre outras características, esse componente oferece uma linguagem intermediária para a composição flexível de grafos paralelos e distribuídos de fluxos de dados, um conjunto de operações típicas pré-definidas e um amplo conjunto de adaptadores que facilitam a utilização de diferentes formatos de fluxos de dados existentes, como XML e sistemas de arquivos.

InfoSphere oferece ainda segurança e confidencialidade para informações compartilhadas, além de ser adequado para *clusters* de qualquer tamanho. Todas essas funcionalidades fazem com que essa plataforma seja altamente dinâmica, sendo utilizada na indústria em diversos cenários, como medicina, astronomia e mercado financeiro.

2.2 Reconfiguração Dinâmica de Aplicações Distribuídas

Mecanismos de reconfiguração dinâmica têm como objetivo permitir que um sistema evolua de uma determinada configuração para outra em tempo de execução, ao contrário de evoluções realizadas em tempo de projeto, caracterizadas por serem estáticas. Em um cenário ideal, uma reconfiguração não deve ter impacto sobre o desempenho de um sistema em execução, o que é muito difícil de se obter. Na prática, os mecanismos de reconfiguração dinâmica são projetados de maneira que não tenham um impacto significativo sobre o sistema, tornando sua utilização viável.

No contexto de aplicações distribuídas, mecanismos de reconfiguração dinâmica possibilitam mudanças na granularidade de paralelismo e na alocação de processos aos recursos disponíveis no ambiente. No primeiro caso, as aplicações podem ser modificadas em tempo de execução por meio da criação ou remoção de processos que as compõem, aumentando ou diminuindo a capacidade de processamento de acordo com a demanda em determinado momento. No caso da alocação de processos, um mecanismo de reconfiguração dinâmica pode ser utilizado para permitir a migração de processos entre máquinas de execução com o intuito de adaptar a aplicação a ambientes com características dinâmicas ou contornar situações de falhas.

Motivados pela natureza dinâmica dos ambientes computacionais modernos e aplicações distribuídas, os mecanismos de reconfiguração dinâmica estão se tornando cada vez mais populares, figurando como uma solução apropriada para as demandas de desempenho e flexibilidade existentes. As principais estratégias atualmente utilizadas para promover reconfiguração dinâmica de aplicações distribuídas envolvem virtualização, balanceamento dinâmico de carga e técnicas de *checkpoint/restart*.

Utrera et al. [2004] propõem o que chamam de maleabilidade virtual para aplicações paralelas baseadas em troca de mensagens. É utilizada uma estratégia de alocação de processadores denominada *Folding by JobType* (FJT) que permite a adaptação de aplicações MPI (*Message Passing Interface*) a mudanças de carga de processamento. A técnica é baseada nos conceitos de dobramento [McCann & Zahorjan, 1994] e moldabilidade [Feitelson et al., 1997]. Árvores de decisão são utilizadas para o estabelecimento do número inicial ótimo de processos, o momento de dobramento de aplicações e o número de "dobras" a serem utilizadas. Todas essas decisões são tomadas com base na observação do comportamento atual e passado da aplicação. As tarefas MPI são classificadas em dois grupos, de execução longa e de execução curta. Quando uma tarefa é submetida ao ambiente, o mecanismo verifica o número de processadores disponíveis e o número de processadores requeridos pela tarefa. Dependendo do tipo de tarefa, decide-se por iniciar sua execução, mesmo que não haja processadores suficientes no momento. Isso é feito por meio do "dobramento" da tarefa, ou seja, a tarefa executa por determinado tempo em menos processadores que precisa, até que outros recursos sejam liberados e alocados a ela. Uma tarefa pode ser dobrada e desdobrada ao longo de sua execução a critério do mecanismo de decisão proposto. Como pode ser observado, a maleabilidade é apenas simulada no ambiente, não ocorrendo, de fato, expansão dinâmica das aplicações.

Huang et al. [2003] apresentam uma implementação MPI denominada *Adaptive MPI* (AMPI) construída sobre o sistema de execução Charm++ [Kale & Krishnan, 1993]. AMPI foi construída no paradigma de orientação a objetos e é composta por uma biblioteca paralela que realiza a migração de objetos de aplicação. Tirando proveito do balanceamento dinâmico de carga e características de portabilidade presentes em Charm++, as reconfigurações são feitas por meio da inicialização das aplicações utilizando alta granularidade de processos, fazendo uso do balanceamento dinâmico de carga para realizar o remapeamento de processos nos recursos físicos de acordo com a demanda medida. O remapeamento é feito por meio da migração de objetos, que por sua vez é provida por um sistema de virtualização presente no ambiente de execução, o que torna o processo de remapeamento mais fácil. Com a estratégia adotada, AMPI não promove a remoção ou adição de novos processos às aplicações, não lidando assim

com os problemas de reparticionamento de estados e redirecionamento de mensagens futuras. Um ponto negativo dessa abordagem está no fato de que o número de processos ociosos pode ser suficientemente grande a ponto de causar um impacto negativo sobre o desempenho das aplicações.

Phoenix [Taura et al., 2003a,b] é outro trabalho que utiliza uma estratégia semelhante de balanceamento de carga para promover reconfiguração dinâmica de aplicações distribuídas. Naquele trabalho é definido um modelo de programação para o desenvolvimento de aplicações paralelas e distribuídas que acomodam mudanças dinâmicas no conjunto de recursos de processamento, isto é, acomodam a criação ou remoção de nós físicos. No modelo proposto os nós físicos envolvidos na computação de uma aplicação têm uma visão de um espaço de nomes virtuais grande e fixo, que tem a função de mapear os processos que compõem a aplicação. A estratégia para promoção de reconfiguração é muito semelhante àquela utilizada em AMPI. Cada nó físico é responsável por um subconjunto dos nós virtuais (processos) de uma determinada aplicação, sendo que cada nó virtual é de responsabilidade de um, e apenas um nó físico. Quando há a necessidade de expandir a aplicação em resposta à disponibilidade de novos nós físicos, é feito um rebalanceamento de carga, o que provoca a migração de nós virtuais dos nós físicos já existente para os novos nós físicos. No caso da contração de uma aplicação, os nós virtuais são migrados no sentido contrário. O processo de migração é realizado em exclusão mútua, o que faz com que as mensagens direcionadas aos nós físicos participantes sejam enfileirados para envio posterior.

Sievert & Casanova [2004] utilizam uma técnica de troca de processos orientada a monitoração de carga de trabalho. Essa técnica tem como objetivo melhorar o desempenho de aplicações MPI em ambientes distribuídos compartilhados. No início da execução de uma aplicação, o sistema cria M processos, sendo $M > N$, onde N é o número de processos solicitado pelo usuário. Depois disso, escolhe um subconjunto inicial com N processos para comporem a aplicação. Os demais $M - N$ processos permanecem inativos em máquinas diferentes das máquinas executando a aplicação. Durante a execução, o sistema periodicamente verifica o desempenho do conjunto de máquinas alocadas à execução de suas aplicações. De acordo com os dados obtidos na monitoração, a computação é movida entre os processos executando em processadores mais lentos para processos executando em processadores mais rápidos. Uma decisão de troca sempre envolve dois processos, um ativo e um inativo. O primeiro envia seus dados e informações de controle para o segundo, tornando-se inativo ao fim da troca. O segundo assume os dados e a computação do primeiro processo, tornando-se ativo na máquina em que foi originalmente criado. O mais interessante desse trabalho é que o usuário não precisa identificar/prover o estado de cada processo, pois toda a memória

dinamicamente alocada por um processo é automaticamente registrada e transferida em uma situação de troca. Além disso, o usuário conta com funções dedicadas para a alocação de memória que não deve ser migrada. Essas características são importantes porque facilitam a utilização do mecanismo, tornando ainda mais alto o nível de abstração oferecido.

Grande parte das abordagens baseadas em balanceamento dinâmico de carga envolvem a criação de um número de processos maior do que necessário para permitir reconfigurações dinâmicas. Essa estratégia pode ter impacto negativo à medida que a manutenção desses processos, sejam eles ativos ou não, torna-se uma tarefa significativa do ponto de vista de desempenho. Uma alternativa a essa estratégia é possibilitar que os elementos de processamento possam ser divididos ou colapsados em tempo de execução, sem prejuízos para a semântica da aplicação. O projeto PCM (*Process Checkpointing and Migration*) [Desell et al., 2006, 2007; El Maghraoui et al., 2007, 2009] consiste em uma biblioteca a nível de usuário que permite que os processos de uma aplicação possam ser divididos ou colapsados, o que faz com que não haja processos inativos ocupando recursos da infraestrutura física. Além disso, processos podem ser migrados entre máquinas de acordo com dados obtidos na monitoração dos recursos. Algumas das tarefas são executadas de forma transparente, sendo que o ambiente toma as decisões de quando reconfigurar as aplicações. Todavia, o programador deve definir como essas transformações devem ser realizadas e como os dados dos processos devem ser redistribuídos após uma reconfiguração.

Schneider et al. [2009] apresentam uma abordagem de elasticidade de operadores para a linguagem de programação do ambiente IBM Infosphere. Esses operadores devem representar funções puras, sem conter estado de computação assinalado a cada um deles e as operações realizadas devem ser associativas e comutativas. Essas características facilitam a tarefa de reconfiguração, uma vez que não há a necessidade de redistribuição de estados ou manutenção de semântica na entrega de mensagens. As mensagens recebidas por determinado operador são colocadas em uma fila global de trabalho. Cada cópia do operador retira mensagens dessa fila para executar o processamento conforme sua capacidade. Um módulo dedicado verifica periodicamente o desempenho instantâneo desse operador, utilizando a taxa de processamento de mensagens como métrica de decisão. O sistema então pode decidir por aumentar ou diminuir a granularidade de paralelismo de dados no operador tendo em vista o aumento da taxa de processamento global de mensagens.

O ambiente Starfish [Agbaria & Friedman, 1999, 2002] é um ambiente de execução de programas MPI estáticos e dinâmicos. A arquitetura do ambiente é muito flexível e permite a implementação de diferentes protocolos de *checkpoint/restart*, se-

jam eles coordenados ou não, podendo executar em conjunto para aplicações diferentes. Independente do protocolo utilizado, os estados dos processos em execução são armazenados em meio persistente para a provisão de mudanças dinâmicas. Essas mudanças incluem adição ou remoção de nós de processamento e situações de falhas ou recuperação. Quando uma mudança é detectada, o que é feito por meio de um mecanismo dedicado a essa tarefa, os processos afetados têm seus últimos estados válidos migrados entre os nós do *cluster*, de acordo com políticas definidas pelo programador de aplicação. Depois de realizada a migração, os processos são reiniciados em seus novos nós de processamento. A migração é a única maneira de se alterar a estrutura de uma aplicação em execução, o que pode ser visto como um ponto negativo dessa abordagem, uma vez que não há como aumentar ou diminuir o número de processos de determinada aplicação.

Vadhiyar & Dongarra [2003] propõem um arcabouço para o desenvolvimento de aplicações distribuídas baseadas em troca de mensagens com assistência a maleabilidade e migração de processos. O arcabouço contém uma biblioteca de *checkpointing* a nível de usuário chamada SRS (*Stop Restart Software*) e um ambiente de execução para a gerência do armazenamento de dados em uma infraestrutura distribuída. A proposta não apresenta alto grau de transparência, uma vez que os usuários devem inserir chamadas em suas aplicações para informar quais dados devem ser armazenados e qual a periodicidade com que a etapa de *checkpointing* deve ser realizada. No caso de uma reconfiguração, cada processo da aplicação tem acesso aos dados armazenados seguindo alguma política de distribuição, também fornecida pelo usuário. Os autores argumentam sobre a contribuição de portabilidade do mecanismo, podendo as aplicações serem executadas em ambientes heterogêneos. No entanto, o mecanismo adiciona uma sobrecarga de 15% a 35% no desempenho das aplicações, o que pode ser proibitivo de acordo com o cenário onde o sistema está sendo empregado.

2.3 Considerações Finais

Nesse capítulo foram apresentados, inicialmente, alguns dos trabalhos de maior evidência na área de processamento distribuído de fluxos de dados. Esses trabalhos são motivados principalmente pela demanda por computação de alto desempenho, processamento contínuo de dados e provisão de abstrações de alto nível para o desenvolvimento de aplicações distribuídas. Como resultado desses esforços, vários ambientes de processamento foram propostos, incluindo um conjunto diversificado de tecnologias e abordagens. Embora esses ambientes compartilhem alguns objetivos, geralmente eles

apresentam diferenças significativas na arquitetura, modelo de dados, padrão de comunicação, modelo de programação e garantias de disponibilidade e escalabilidade.

O ambiente Watershed, descrito em detalhes no capítulo 3, foi projetado para oferecer diversos mecanismos de apoio às características dinâmicas das aplicações de processamento distribuído de fluxos de dados. Dentre essas características encontram-se a carga e remoção de elementos de processamento em tempo de execução, a elasticidade das etapas de processamento (aumento ou redução do grau de paralelismo) e a coexistência de aplicações distintas compartilhando resultados entre si. Essas funcionalidades situam o ambiente como um projeto diferenciado dos trabalhos apresentados nesse capítulo, uma vez que nenhuma proposta estudada integra todas as funcionalidades oferecidas pelo Watershed.

Um dos grandes desafios que impulsionam o aprimoramento desses ambientes é a capacidade de lidar com mudanças dinâmicas na disponibilidade e demanda por processamento. Essas mudanças podem ocorrer na infraestrutura física que abriga a execução das aplicações ou na carga de trabalho imposta às aplicações, que apresentam uma natureza dinâmica acentuada, podendo exigir mais ou menos recursos computacionais ao longo do tempo. Na segunda parte desse capítulo foram discutidos alguns trabalhos que apresentam soluções para permitir a reconfiguração dinâmica de aplicações distribuídas. Grande parte dos trabalhos utilizam estratégias de balanceamento de carga, virtualização de processadores e migração de processos, separadamente ou em conjunto para aumentarem ou diminuírem a granularidade do paralelismo de dados das aplicações, reagindo às diversas mudanças observadas no cenário de execução.

Algumas das estratégias apresentadas exigem grande esforço do programador de aplicação na definição das operações de reconfiguração, sendo o mesmo responsável pela redistribuição de dados e definição da forma como os processos são adicionados ou removidos de suas aplicações. Outras contornam o problema de redistribuição de dados utilizando um número expressivo de processos, alternando a computação entre eles, incorrendo na sobrecarga da tarefa de gerenciamento de processos.

A proposta desse trabalho inclui a descrição de um mecanismo de assistência eficiente a mudanças dinâmicas na topologia de aplicações de processamento contínuo. O mecanismo apresenta alto grau de abstração, sendo praticamente transparente ao programador de aplicação. Como plataforma básica de execução, é também apresentado um novo ambiente de processamento distribuído de fluxos de dados, baseado no modelo filtro-fluxo e estritamente orientado a dados.

Capítulo 3

O Ambiente Watershed

Nesse capítulo será apresentado o ambiente de processamento distribuído de fluxos de dados denominado Watershed. Será feita uma descrição das principais características do ambiente, incluindo o modelo de programação utilizado, sua arquitetura, o funcionamento das operações por ele realizadas e alguns resultados experimentais.

3.1 Visão Geral

Watershed [Ramos et al., 2011] é um arcabouço para o desenvolvimento de aplicações distribuídas, projetado para permitir a análise *online* de grandes volumes de dados. Os principais objetivos desse ambiente incluem a provisão de abstrações de alto nível, que facilitem o desenvolvimento de aplicações distribuídas e a utilização eficiente dos recursos computacionais disponíveis para melhorar o desempenho das mesmas.

Aplicações para o ambiente Watershed são desenvolvidas no modelo de processamento filtro-fluxo. Nesse modelo, uma aplicação é composta por uma cadeia de elementos de processamento, denominados filtros, que se comunicam por meio de fluxos contínuos de dados. Cada filtro representa uma etapa do processamento feito pela aplicação e um fluxo de dados é composto por resultados intermediários de alguma etapa de processamento que são insumos para etapas posteriores. O encadeamento de filtros possibilita o paralelismo de tarefas, uma vez que a computação pode ser desmembrada em quantas etapas forem convenientes. O paralelismo de dados é obtido por meio da replicação de um filtro em um conjunto de instâncias idênticas que podem executar em máquinas diferentes de um *cluster*, sendo que os dados de entrada do filtro replicado são distribuídos entre as instâncias criadas. Tal modelo é ilustrado na figura 3.1. Em Watershed, os processos de encadeamento e replicação de filtros são realizados de maneira transparente, como será detalhado mais adiante.

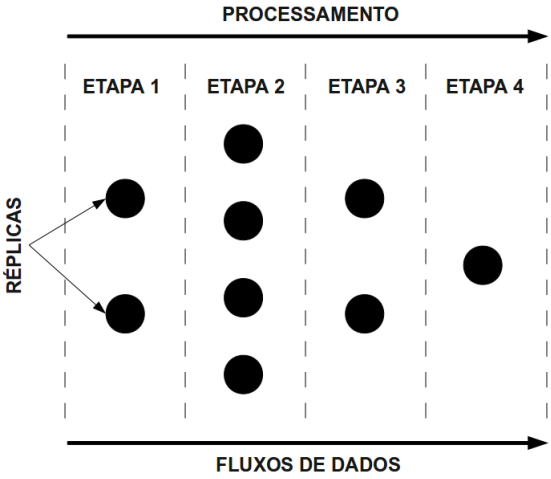


Figura 3.1: Composição de processamento no modelo filtro-fluxo. Aplicação dividida em várias etapas com replicação interna, obtendo-se paralelismo de tarefas e paralelismo de dados.

As abstrações oferecidas por Watershed tornam fácil a tarefa de criação de aplicações distribuídas voltadas para o processamento de fluxos de dados. Os desenvolvedores concentram seus esforços na programação dos filtros de processamento e na definição dos tipos de dados que consomem e que produzem. É responsabilidade do ambiente estabelecer a conexão entre as instâncias dos filtros, administrar o roteamento de mensagens e coordenar a execução como um todo.

As conexões entre filtros são realizadas em tempo de execução pelo ambiente e são orientadas estritamente a dados, sendo que um filtro de processamento pode consumir dados de zero ou mais fluxos e pode produzir dados para no máximo um fluxo. A existência de um fluxo de dados está condicionada à existência de pelo menos um filtro produtor de dados para o mesmo. Dessa forma, filtros consumidores de algum tipo de dados ainda não produzido no ambiente no momento de sua carga permanecem ociosos até que os mesmos comecem a ser gerados por algum filtro adicionado ao ambiente posteriormente.

A conexão orientada a dados diz respeito ao fato de que se um filtro F tem como entrada os dados de um fluxo S, todos os filtros que produzem dados para S são dinamicamente conectados a F como produtores, independentemente do momento de carga de cada filtro envolvido. Quanto ao roteamento das mensagens, um filtro deve definir, para cada fluxo de entrada, como os dados devem ser entregues às suas instâncias de acordo com uma das políticas de distribuição disponíveis (*Broadcast*, *Round-robin* e *Fluxo Rotulado*), que serão detalhadas na seção 3.2.3.

As aplicações em Watershed, diferentemente de grande parte dos ambientes pro-

postos na literatura, executam em modo contínuo, sem o conceito de terminação. Em outras palavras, os filtros de processamento executam continuamente até que o usuário decida removê-lo por meio de uma chamada explícita a uma função de terminação que faz parte da API (*Application Programming Interface*) do ambiente ou por meio de um comando externo disparado do *console* de interação do ambiente.

Embora a literatura apresente propostas interessantes de ambientes de processamento de fluxos de dados, Watershed oferece um conjunto de funcionalidades que o torna apropriado para a tarefa de processamento contínuo de grandes volumes de dados. A primeira funcionalidade diz respeito ao mecanismo de adição e remoção de filtros de processamento em tempo de execução. Essa funcionalidade permite que múltiplas aplicações executem simultaneamente, compartilhando resultados intermediários para algum processamento específico, ou cooperando para um propósito comum. Outra vantagem de tal mecanismo é a possibilidade da realização de alterações na topologia das aplicações em tempo de execução, o que é conveniente em cenários onde novas etapas de processamento sejam demandadas ou não mais desejadas a partir de um dado momento da execução. Adicionalmente, o reúso dos filtros em execução implica em ganho de desempenho, uma vez que elimina computações repetidas sobre os mesmos dados.

Outra funcionalidade importante diz respeito ao módulo de persistência de fluxos. Esse módulo é responsável por persistir todos os dados produzidos no ambiente, criando um grande repositório que permite a um filtro processar dados produzidos antes de sua carga no sistema, além de possibilitar o rastreamento das transformações realizadas nos dados desde a entrada dos mesmos no ambiente.

Uma terceira funcionalidade inclui um modelo dinâmico de previsão e balanceamento de carga, apoiado por um mecanismo de manutenção de estados dos filtros em execução. Esses dois componentes funcionam de forma cooperativa com o objetivo de melhorar a utilização dos recursos computacionais disponíveis e aumentar a eficiência das aplicações em execução. Em termos gerais, o modelo de previsão tem como insumo alguns dados de execução que possibilitam a tomada de decisões sobre a configuração dos filtros em execução, que incluem a alteração da localização e do número de instâncias dos mesmos. O módulo de manutenção de estados é responsável por possibilitar alterações topológicas e estruturais dos processos que compõem os filtros. O grande desafio desse último componente é a manutenção da consistência do ambiente, que pode ser vista como a possibilidade de criação, remoção e migração de instâncias de filtros sem prejuízos para a computação final em termos de dados consumidos e produzidos.

As funcionalidades oferecidas pelo ambiente Watershed o fazem um ambiente dinâmico e flexível, atendendo à demanda crescente por aplicações eficientes para o

processamento *online* de grandes volumes de dados.

3.2 Arquitetura

A arquitetura do ambiente Watershed é logicamente composta por quatro camadas, denominadas *Aplicação*, *Controle*, *Comunicação* e *Persistência*. A figura 3.2 apresenta a organização em camadas, juntamente com os principais componentes e funções de cada uma delas.

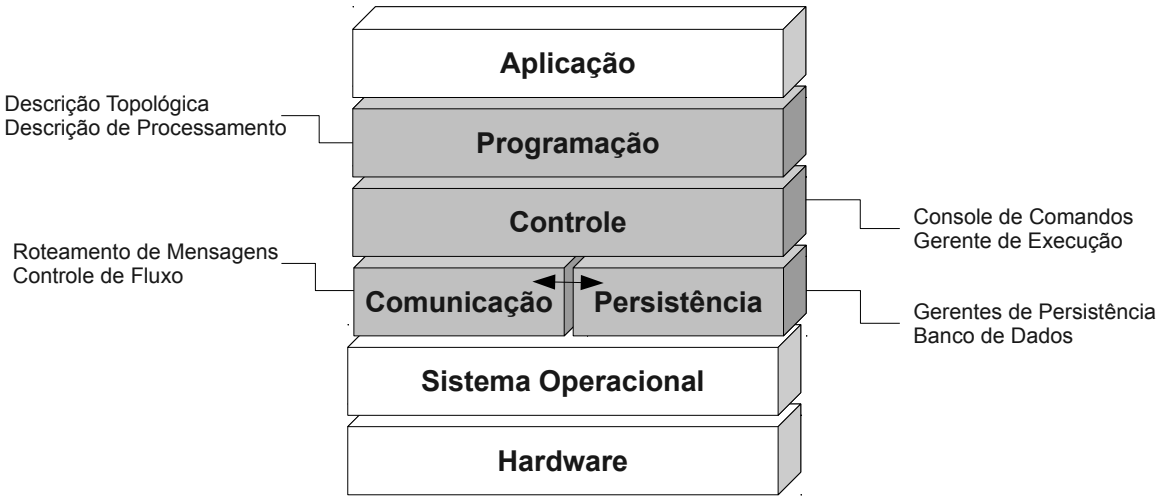


Figura 3.2: Arquitetura em camadas do ambiente de processamento Watershed e seus principais componentes.

A camada de programação estabelece o formato de criação de um filtro de processamento e provê uma API que inclui as funções exportados pelo ambiente para utilização pelos filtros. Os componentes da camada de controle são responsáveis pela carga e descarga dos filtros de processamento e pelo controle de toda a execução dos mesmos, coordenando todos os mecanismos associados às funcionalidades do ambiente. A camada de comunicação cuida da troca de mensagens entre todos os processos em execução no ambiente, incluindo filtros de processamento e os componentes de gerência. Por fim, a camada de persistência gerencia a persistência de todos os fluxos de dados ativos no ambiente e o consumo de dados persistidos.

3.2.1 Camada de Programação

Uma aplicação projetada para executar em Watershed é composta por duas partes: uma descrição topológica e uma descrição de processamento. A topologia de

uma aplicação é descrita de maneira declarativa, por meio de um conjunto de arquivos XML de configuração, um para cada filtro de processamento. Como mencionado anteriormente, o usuário não declara explicitamente as conexões entre os filtros de processamento, sendo necessário apenas explicitar os fluxos dos quais o filtro consome dados e o fluxo para o qual produz dados, sendo as conexões realizadas pelo ambiente. A figura 3.3 apresenta o DTD (*Document Type Definition*) de definição e validação da estrutura de configuração de um filtro e um arquivo XML contendo um exemplo de configuração.
O arquivo de configuração de um filtro contém os seguintes campos:

- `name`: elemento obrigatório – identificador unívoco do filtro de processamento.
- `library`: elemento obrigatório – localização absoluta da biblioteca dinâmica que contém o código de processamento do filtro.
- `instances`: elemento opcional – define o número de instâncias do filtro. Quando ausente, o ambiente define um número arbitrário de instâncias de acordo com a disponibilidade de recursos.
- `arguments`: elemento opcional – utilizado para passagem de argumentos para as instâncias do filtro.
- `inputs`: elemento opcional – lista de fluxos dos quais o filtro consome dados. Cada item dessa lista deve conter o nome do fluxo de entrada e a política (`policy`) de distribuição de mensagens.
- `output`: elemento opcional – nome do fluxo para o qual o filtro produz dados.
- `demands`: elemento opcional – lista de demandas para execução do filtro. Esse elemento tem como finalidade a atribuição das instâncias do filtro a máquinas que tenham os recursos demandados. Um exemplo de utilização desse campo é definir que determinado filtro necessita executar em máquinas que tenham acesso à Internet. Dessa forma, o ambiente de execução irá escalonar as instâncias de tal filtro para máquinas que tenham essa característica.

A descrição de processamento de um filtro, por sua vez, é feita de forma procedural. Watershed provê uma API, atualmente disponível em C++, que contém os métodos básicos para a implementação de filtros de aplicação. Novos filtros são criados como classes C++ especializadas e disponibilizadas pelo usuário na forma de bibliotecas dinâmicas. Em uma classe de filtro, o programador deve implementar um pequeno

```
<!ELEMENT filter (global, inputs?, output?, demands?)>
  <!ELEMENT global (#PCDATA)>
    <!ATTLIST global
      name CDATA #REQUIRED
      library CDATA #REQUIRED
      instances CDATA #IMPLIED
      arguments CDATA #IMPLIED>
  <!ELEMENT inputs (input+)>
    <!ELEMENT input (#PCDATA)>
      <!ATTLIST input
        name CDATA #REQUIRED
        policy (broadcast|round_robin|labeled) "round_robin"
  <!ELEMENT output (#PCDATA)>
    <!ATTLIST output
      name CDATA #REQUIRED
  <!ELEMENT demands (demand+)>
    <!ELEMENT demand (#PCDATA)>
      <!ATTLIST demand
        name CDATA #REQUIRED>
```

(a)

```
<filter>
  <global name="FiltroExemplo"
    library="/home/ws/FiltroExemplo.so"
    instances="3"
    arguments="-i entrada.txt -o saida.txt">
  </global>
  <inputs>
    <input name="Fluxo1" policy="labeled"/>
    <input name="Fluxo2" policy = "broadcast"/>
    <input name="Fluxo3" policy = "round_robin/>
  </inputs>
  <output name="FluxoExemplo"> </output>
  <demands>
    <demand name="MySQL"/>
    <demand name="AcessoWeb"/>
  </demands>
</filter>
```

(b)

Figura 3.3: Descrição topológica de um filtro de processamento em Watershed. Em (a) é mostrado o arquivo que define e valida a estrutura de uma configuração de filtro. Em (b) é ilustrado um exemplo de um arquivo de configuração.

conjunto de métodos que são invocados pelo ambiente para o tratamento de ações específicas, a saber:

- `void Process(Message&)`: método de processamento do filtro. O ambiente Wa-

tershed invoca esse método sempre que há uma mensagem de dados proveniente de alguma instância de filtro produtor.

- `int GetLabel(Message&)`: determina um rótulo para determinada mensagem de dados. Por meio desse rótulo, a mensagem é direcionada a uma das instâncias do filtro destino utilizando a política de fluxo rotulado. A implementação do usuário recebe como argumento a mensagem a ser rotulada e deve retornar o rótulo atribuído a essa mensagem.

Fazem parte da API do ambiente alguns métodos que podem ser invocados pelo usuário durante a execução do filtro:

- `string GetArgument(string)`: esse método recupera o valor de um argumento passado ao filtro com base no identificador que procede o argumento no arquivo de configuração.
- `string GetName(void)`: retorna o identificador textual unívoco do filtro.
- `int GetNumberInstances(void)`: retorna o número de instâncias do filtro.
- `int GetRank(void)`: retorna o identificador numérico da instância chamadora.
- `void Send(Message&)`: escreve uma mensagem de dados no fluxo produzido pelo filtro.
- `void SyncConsumers(Message&)`: envia uma mensagem de sincronização para todas as instâncias de todos os filtros que consomem dados do fluxo de saída do filtro. A instância que chama o método permanece bloqueada nesse método até que todos os consumidores tenham recebido a mensagem.
- `void Terminate(void)`: termina a execução da instância que a executa.

Um objeto do tipo `Message` contém alguns métodos que podem ser invocados pelo código do usuário:

- `void* GetData(void)`: retorna um ponteiro para os dados contidos dentro de uma mensagem.
- `int GetDataSize(void)`: retorna o tamanho em *bytes* dos dados contidos na mensagem.
- `int GetSource(void)`: retorna o identificador (*rank*) da instância onde foi produzida a mensagem.

- `string GetStream(void)`: retorna o identificador textual do fluxo de onde a mensagem foi recebida.
- `int GetTimestamp(void)`: retorna o tempo de criação da mensagem na instância do filtro produtor.
- `void SetData(void*, int)`: atribui dados de usuário à mensagem. Recebe como parâmetros um ponteiro para uma área de memória e a quantidade de *bytes* a serem copiados para a mensagem.

A figura 3.4 mostra um exemplo de código de processamento de um filtro. Nesse exemplo simples, o filtro recebe números inteiros de um fluxo de entrada e escreve a média atual desses números no fluxo de saída.

```
#include <watershed.h>

class FiltroExemplo: public Filter {
  public:
    FiltroExemplo::FiltroExemplo(void) {
      n_numeros = 0;    soma = 0;    media = 0.0;
    }

    FiltroExemplo::~FiltroExemplo(void) {
      std::cout << "Media final em " << GetName( ) << GetRank( ) << ": " << ↩
          media << endl;
    }

    void FiltroExemplo::Process(Message& M) {
      int* N = (int*) M.GetData( );
      n_numeros = n_numeros + 1;    soma = soma + *N;
      media = (float) soma / n_numeros;
      Message MS;
      MS.SetData(&media, sizeof(float));
      Send(MS);
    }

  private:
    int n_numeros, soma;
    float media;
}
```

Figura 3.4: Exemplo de descrição de processamento de um filtro Watershed.

Como descrito na presente seção, os processos de criação de um filtro Watershed e a composição de aplicações distribuídas são muito intuitivos, uma vez que o ambiente provê abstrações de alto nível convenientes a essas tarefas. As funcionalidades ofereci-

das pelo ambiente encapsulam toda a gerência de processos, roteamento de mensagens e controle de execução, desonerando o programador dessas responsabilidades.

3.2.2 Camada de Controle

A camada de controle tem como função coordenar a execução de todos os filtros de processamento. Nessa camada são implementadas as funcionalidades de controle de execução de filtros de processamento, mudança dinâmica de topologia de aplicação e balanceamento de carga. A camada é formada por dois componentes principais, o ***console* de comandos** do ambiente e o **gerente de execução**.

O *console* é um programa por meio do qual usuários podem interagir com o ambiente. Os usuários podem iniciar o ambiente, realizar consultas de fluxos de dados ativos, invocar rotinas de mudanças topológicas em aplicações, adicionar filtros de processamento e removê-los, quando necessário. Além disso, possui um comando para finalizar a execução do ambiente, que tem como ação a finalização de todos os filtros em execução, seguida da finalização dos componentes do ambiente. A interação entre usuários e o Watershed é ilustrada na figura 3.5.

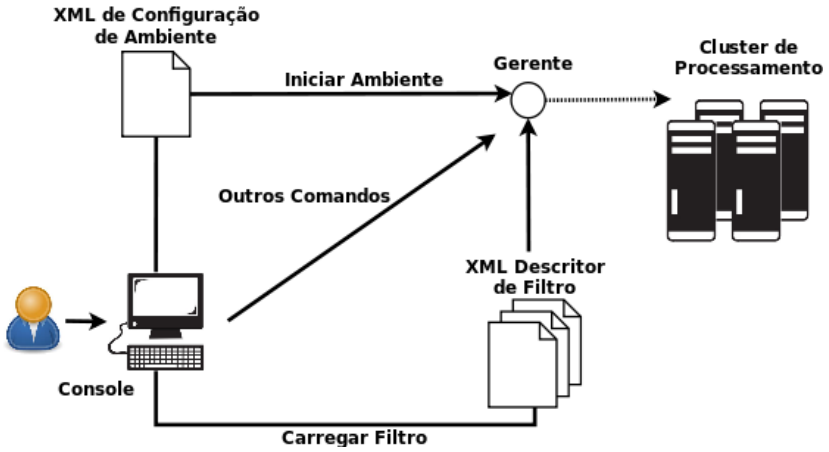


Figura 3.5: Diagrama de operação do *console* de ambiente.

Para cada comando, o *console* interage com os outros componentes do ambiente, exibindo mensagens contendo informações sobre a execução da operação solicitada pelo usuário. Para iniciar o sistema, um usuário deve disparar um comando específico acompanhado do arquivo XML de configuração, conforme mostrado na figura 3.6. Esse arquivo descreve a estrutura do ambiente, que inclui informações de configuração, lista de máquinas disponíveis e os recursos ofertados por cada uma delas. Usando esse arquivo de configuração, os componentes do Watershed são iniciados no *cluster*, fazendo com que o ambiente esteja pronto para executar aplicações.

```
<config>
  <global>
    <ompi prefix="/opt/ompi-1.5.4">
    </ompi>
    <server name="ws-manager"
      home="/home/libws"
      running_dir="/var/tmp">
    </server>
  </global>
  <hostdec>
    <host name="host1" database_server="true">
      <resource name="R1"/>
      <resource name="R2"/>
    </host>
    <host name="host2" database_server="false">
      <resource name="R3"/>
    </host>
  </hostdec>
</config>
```

Figura 3.6: Arquivo XML de configuração de ambiente para execução.

Iniciado o sistema, os usuários podem então interagir com o mesmo por meio dos comandos do *console*. Os principais comandos existentes são:

- `start`: carrega o ambiente Watershed no *cluster* de processamento.
- `load-filter [xml_path]`: comando para adicionar um novo filtro a alguma cadeia de processamento em execução. É necessário prover o arquivo XML de configuração contendo as informações sobre o filtro, como explicado anteriormente.
- `unload-filter [filter_name]`: ativa o processo de remoção de um filtro que é feito por meio do nome identificador do filtro, passado como argumento do comando.
- `list-active-streams`: lista todos os fluxos de dados ativos no ambiente.
- `list-streams`: lista fluxos de dados ativos e fluxos de dados persistidos, informando a distribuição dos dados persistidos entre as instâncias de banco de dados existentes.
- `stop`: remove todos os filtros em execução e termina todos os processos gerentes.

O gerente de execução controla a execução de todos os filtros ativos, além de ser responsável por efetuar a carga e a remoção dos mesmos, quando solicitado pelo usuário.

Na fase de carga de um filtro, o gerente deve distribuir as instâncias do mesmo entre as máquinas disponíveis. A decisão de escalonamento é feita com base nas demandas declaradas para cada filtro e o número de instâncias solicitadas. As demandas definem quais máquinas são elegíveis para executar instâncias do filtro em questão e o número de instâncias refere-se ao número de réplicas a serem executadas. Dado o conjunto de máquinas elegíveis, o gerente distribui as instâncias do filtro entre as máquinas do conjunto, utilizando a ordem inversa da ocupação das mesmas por instâncias já em execução.

Durante a execução de um filtro, o gerente provê informações sobre como o filtro deve escrever dados no fluxo de saída e ler dados dos fluxos de entrada. Em termos práticos, o gerente informa como o filtro deve se conectar aos gerentes de persistência, aos seus filtros produtores e aos seus filtros consumidores. Após estabelecer todas as conexões necessárias, o filtro começa a enviar e receber dados. Além disso, o gerente é responsável por enviar mensagens de controle sobre alterações que afetem a execução de um filtro – a remoção de um produtor, por exemplo.

Quando a remoção de um filtro é solicitada, o gerente de execução envia uma mensagem de notificação para todos os produtores e consumidores do filtro a ser removido. Em seguida, envia uma mensagem de terminação para todas as instâncias do referido filtro, que, de maneira síncrona, desconectam-se de seus produtores e consumidores.

Outras atribuições do gerente incluem as decisões de balanceamento de carga e alterações topológicas dos filtros em execução. O balanceamento de carga é feito com base na monitoração periódica do ambiente e na utilização de um modelo probabilístico para a tomada de decisões sobre a topologia dos filtros em execução. Essa funcionalidade não será abordada no presente trabalho por estar em fase experimental. Para viabilizar alterações na topologia dos filtros em execução foi desenvolvido um mecanismo de adição, remoção e migração de instâncias de filtros em tempo de execução. O mecanismo é baseado na estratégia de *checkpoint-restart* e será detalhado no capítulo 4.

3.2.3 Camada de Comunicação

A camada de comunicação tem como principais responsabilidades definir o protocolo de comunicação entre todos os processos em execução no ambiente Watershed e assegurar a entrega correta dos dados trocados entre eles. Esses processos incluem os componentes do ambiente e as instâncias de filtros de processamento.

A comunicação entre processos em Watershed é feita por meio de troca de mensagens, utilizando-se o padrão MPI. A escolha de MPI foi feita porque o mesmo possui algumas vantagens desejáveis sobre outras plataformas de passagem de mensagens,

destacando-se a maturidade do padrão, as boas implementações existentes e alto grau de portabilidade.

Watershed tem uma camada de comunicação construída sobre a biblioteca Open MPI [Gabriel et al., 2004], que fornece as funções básicas de envio e recebimento de mensagens, bem como funções para o gerenciamento de processos. Para possibilitar a utilização do padrão e da biblioteca, foi realizado um mapeamento entre os componentes da arquitetura do ambiente e os conceitos disponíveis em MPI. Todos os gerentes, o *console* de comandos e as instâncias dos filtros de processamento são implementados como processos MPI. Isso é importante para tornar homogênea a forma de comunicação entre todos os processos, o que não seria possível se um dos componentes não fosse implementado como um processo MPI.

O padrão MPI estabelece um mecanismo chamado comunicador que permite definir módulos que encapsulam operações de comunicação interna. Um comunicador identifica um grupo de processos e o contexto em relação ao qual uma determinada operação deve ser efetuada. Processos podem pertencer a um ou mais grupos. Os processos em um grupo são identificados por inteiros, únicos e numerados sequencialmente a partir de zero. Essa identificação é denominada *rank* do processo. Existem dois tipos de comunicadores: o intracomunicador e o intercomunicador. Um intracomunicador é utilizado para a comunicação dentro de um grupo de processos, enquanto um intercomunicador é utilizado para a comunicação entre processos de grupos distintos. O intercomunicador funciona como uma ponte entre dois grupos de processos, onde qualquer processo em um dos grupos envolvidos pode enviar mensagens para qualquer processo do outro grupo.

A organização dos processos em Watershed e a troca de mensagens entre os mesmos são realizadas da seguinte maneira. O *console* do sistema é um programa invocado apenas no momento de sua utilização. Ele pode ser executado a partir de qualquer máquina do *cluster* onde Watershed está instalado e comunica-se apenas com o gerente de execução para enviar comandos dados pelo usuário. Essa comunicação segue o modelo cliente-servidor, onde o gerente abre uma porta de comunicação e aguarda conexões do *console* para a troca de mensagens através de um intercomunicador. Após a conclusão de um comando, a conexão é desfeita e o *console* é finalizado.

O gerente de execução é criado e disposto sozinho dentro de um grupo MPI. Após o início de sua execução, o gerente dispara os gerentes de persistência de acordo com o arquivo de configuração e os coloca em um grupo de processos dedicado. A comunicação entre esses gerentes de persistência é feita utilizando-se um intracomunicador com o propósito de trocar informações sobre fluxos de dados ativos no ambiente. Depois de criado, o grupo de gerentes de persistência abre uma porta de conexão para que os

filtros adicionados ao ambiente possam se conectar a eles.

Quando o gerente de execução recebe um comando de carga de um novo filtro, ele dispara as instâncias desse filtro, que também são agrupadas logicamente. A comunicação entre gerente e instâncias de filtros de processamento é feita por meio de um intercomunicador, que conecta os dois grupos envolvidos. O grupo de instâncias do filtro conecta-se ao grupo de gerentes de persistência. Isso é feito para que os dados produzidos pelo novo filtro possam ser persistidos e para que o filtro possa obter dados de fluxos persistidos, caso essa opção esteja presente em sua declaração. Cada filtro de processamento, mapeado em um grupo de processos, abre sua própria porta para receber conexões dos filtros adicionados no futuro. Depois disso, ele recebe do gerente de execução uma lista de portas de seus produtores e consumidores, aos quais ele deve se conectar para iniciar a troca de dados. Realizadas as conexões necessárias, o filtro torna-se parte de uma ou mais cadeias de processamento executando no Watershed.

Como mencionado anteriormente, um filtro declara uma lista de fluxos de dados de entrada e, para cada um, define a política de distribuição de mensagens a ser utilizada. As políticas de distribuição de mensagens disponíveis são:

- *Round Robin* (RR). As mensagens são alternadamente entregues às instâncias do filtro.
- *Broadcast* (BC). As mensagens são entregues a todas as instâncias do filtro.
- *Fluxo Rotulado* (FR). As mensagens são entregues às instâncias do filtro de acordo com uma função de dispersão aplicada aos dados da mensagem. Tal função deve ser implementada pelo programador do filtro.

Independentemente da política de distribuição utilizada, Watershed garante a ordenação parcial das mensagens trocada entre duas instâncias de filtros. Em outras palavras, se um instância de um filtro produz a mensagem m_i no tempo t_i e a mensagem m_j no tempo t_j, sendo $i < j$, todas as instâncias de filtros consumidores que receberem as duas mensagens, as receberão em tempos t_k e t_l, respectivamente, sendo $k < l$.

Um problema na execução de aplicações distribuídas diz respeito ao controle de fluxo, o qual é necessário para não permitir que um consumidor seja sobrecarregado por um produtor mais rápido que ele. Embora Open MPI possua uma estratégia de *buffering* no lado do consumidor, o que poderia ser utilizado para solucionar esse problema, é possível que tais *buffers* cresçam a ponto de tornar a memória principal da máquina insuficiente, causando a interrupção indesejada da execução da aplicação.

Para eliminar esse comportamento, Watershed controla a comunicação entre duas instâncias de filtros por meio de um esquema baseado em créditos [Liu & Panda, 2004].

Para todo consumidor, é definido um número máximo de mensagens que o mesmo pode manter em sua fila de entrada. Esse número é dividido entre as instâncias produtoras. Quando uma instância produtora envia uma mensagem, seus créditos, com relação à instância consumidora, são decrementados. Quando os créditos chegam a zero, a instância produtora envia uma mensagem solicitando uma nova remessa de créditos, ficando impedida de enviar novas mensagens para a instância consumidora até que essa última responda à requisição. Esse processo é ilustrado no algoritmo da figura 3.7.

A implementação do controle de fluxo foi feita de maneira a permitir que as mensagens de requisição de crédito tenham prioridade sobre as mensagens de dados. Dessa forma, caso uma instância produtora envie uma requisição de créditos antes que uma instância consumidora processe todo o *buffer* à primeira reservado, uma oferta de crédito será enviada com o número de mensagens que já foi processado, desbloqueando a instância produtora o mais rápido possível. No caso de alterações topológicas, o compartilhamento de *buffer* e os créditos de todas as instâncias envolvidas são recalculados.

3.2.4 Camada de Persistência

Uma funcionalidade que faz de Watershed um ambiente de processamento de fluxos de dados diferenciado é a existência de um módulo de persistência dos dados produzidos pelos filtros de processamento. A motivação para se manter um módulo dedicado a essa tarefa está no fato de que, para muitas aplicações, pode ser interessante que um filtro processe dados produzidos em tempo real e dados produzidos antes de sua criação, o que não é oferecido pela maioria dos sistemas de processamento de fluxos de dados. Além disso, a persistência é importante para a manutenção do processo de transformação pelo qual os dados são submetidos, oferecendo uma mecanismo de rastreamento, haja vista o fato de que cada fluxo persistido corresponde ao resultado intermediário de uma etapa de processamento.

Os **gerentes de persistência** são responsáveis pela interação com as instâncias de banco de dados instaladas em um subconjunto das máquinas que compõem o *cluster* de processamento, realizando as tarefas de persistência e consultas dos fluxos de dados. Esses gerentes também são responsáveis por manter as informações sobre cada fluxo de dados existente no ambiente, seja ele ativo (sendo produzido por algum filtro), ou persistido (produzido em algum momento – existente apenas no banco de dados). Essas informações são utilizadas no processo de conexão dinâmica entre os filtros, no qual os fluxos de saída dos filtros são casados com os fluxos de entrada dos demais.

A implementação dessa camada faz parte de outro trabalho do grupo de pesquisa,

```
Entrada: M: Mensagem a ser enviada.
Dados: C: Filtros consumidores. CR: Créditos para envio. P: Instâncias pendentes.
início
    P ← ∅
    para cada c ∈ C faça
        I ← SelecionaDestinos(c,M)
        para cada i ∈ I faça
            se CR[c,i] > 0 então
                Envia(M,c,i)
                CR[c,i] ← CR[c,i] - 1
            fim
            senão
                RequisitaCredito(c,i)
                P ← P ∪ {c,i}
            fim
        fim
    fim
    enquanto P ≠ ∅ faça
        {c,i} ← p, p ∈ P
        cr ← EsperaCredito(c,i)
        se cr > 0 então
            CR[c,i] ← cr - 1
            P ← P - {c,i}
            Envia(M,c,i)
        fim
        senão
            RequisitaCredito(c,i)
        fim
    fim
fim
```

(a)

```
Entrada: M: Mensagem recebida.
Dados: CR: Créditos para recebimento.
início
    {p,i} ← RecuperaOrigem(M)
    se M = requisição de crédito então
        cr ← CR[p,i]
        CR[p,i] ← 0
        EnviaCredito(cr,p,i)
    fim
    senão
        Processa(M)
        CR[p,i] ← CR[p,i] + 1
    fim
fim
```

(b)

Figura 3.7: Algoritmo de controle de fluxo baseado em créditos. (a) operação do lado de uma instância produtora. (b) ações simétricas do lado da instância consumidora.

já em fase conclusiva. Por tal motivo, os detalhes de funcionamento e resultados da avaliação experimental não serão abordados no presente trabalho.

3.3 Avaliação Experimental

Essa seção apresenta os resultados da avaliação experimental do ambiente Watershed sem a utilização dos mecanismos de persistência de fluxos e balanceamento de carga. O objetivo desses experimentos é investigar as funcionalidades básicas do ambiente e o desempenho obtido com a paralelização de algumas aplicações executando sobre Watershed. Os experimentos foram conduzidos em um *cluster* com 16 máquinas interconectadas por um *Switch Gigabit Ethernet*. Cada máquina do *cluster* é equipada com um processador Intel ® Core ™ 2 CPU 6420 @2.13GHz, possui 2GB de memória RAM e executa o sistema operacional Linux.

Os experimentos são divididos em duas etapas. Na primeira etapa, foi desenvolvido um conjunto de filtros de processamento que compartilham resultados intermediários na tarefa de análise de transações coletadas da *Web*. Em uma segunda bateria de experimentos, foi avaliado o desempenho de três algoritmos de mineração de dados implementados em Watershed.

3.3.1 Ferramenta de Processamento de Dados da *Web*

A primeira aplicação é composta por um conjunto de filtros simples que fazem análise de dados coletados da *Web*. A fonte de dados escolhida foi o *Twitter*, que é um serviço de rede social *online* que permite aos seus usuários escrever e ler mensagens contendo até 140 caracteres de texto, denominadas *tweets*, compartilhando opiniões, informações de interesse e etc.. O *Twitter* disponibiliza os dados gerados por seus usuários por meio de uma API[1] de fácil utilização, o que permite formar uma base de teste com dados reais de tamanho significativo em pouco tempo de coleta. Para esse experimento foram coletados 10 milhões de *tweets* sobre temas diversos, que foram armazenados em uma base de dados para utilização na ferramenta de processamento desenvolvida.

É importante observar que a ferramenta de processamento proposta nesse experimento poderia atuar diretamente aos dados sendo coletados. No entanto, a taxa com que o *Twitter* disponibiliza os dados não é suficiente para impor carga significativa aos filtros, inviabilizando a realização de experimentos de desempenho.

[1] http://dev.twitter.com/

A figura 3.8 mostra a estrutura da ferramenta de análise, onde é possível identificar o compartilhamento de resultados intermediários por filtros de processamento em aplicações distintas, o que é um benefício direto do mecanismo dinâmico de composição de aplicações. Para essa ferramenta foi utilizada a política de distribuição de mensagens *Round-robin* em todos os filtros, dado o objetivo de distribuir a carga de trabalho igualmente entre as instâncias de cada filtro.

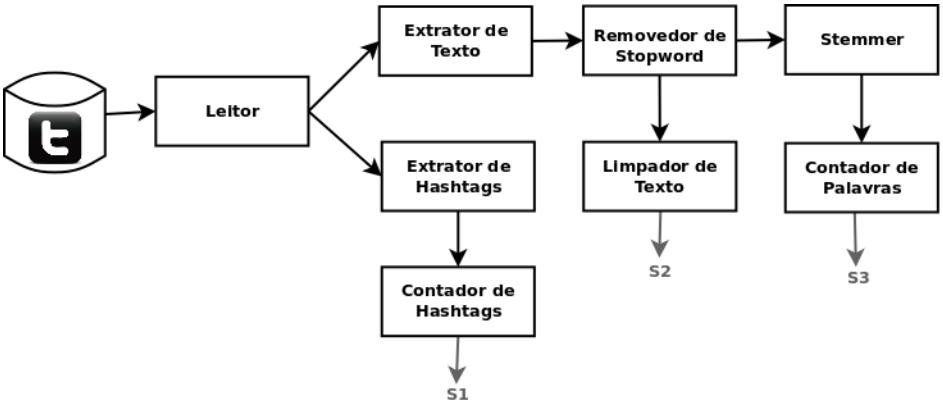


Figura 3.8: Composição de aplicações de processamento de dados do *Twitter*.

Nesse conjunto de aplicações, há um filtro denominado `Leitor`, que realiza a leitura dos dados, no formato de *tweets*, armazenados na base de teste e os escreve em um fluxo de saída que é lido por dois filtros de processamento, `Extrator de Texto` e `Extrator de Hashtags`. Os extratores são responsáveis por extrair o campo de texto e as *hashtags* existentes nos *tweets* coletados, respectivamente. Uma *hashtag*, identificada pelo símbolo #, é usada para marcar uma palavra chave ou um tópico em um *tweet*. Um único *tweet* pode conter zero ou mais *hashtags*.

O `Extrator de Hashtags` escreve em um fluxo de dados consumido pelo `Contador de Hashtags`, cujos dados produzidos podem ser utilizados, por exemplo, para produzir uma visualização das *hashtags* mais utilizadas no momento. O filtro `Extrator de Texto` escreve em um fluxo de dados consumido pelo `Removedor de Stopword`, que é responsável por eliminar palavras com pouco significado para a aplicação, como artigos, preposições, etc.. O texto contendo apenas palavras significativas é então escrito em um fluxo de dados que é consumido por dois outros filtros, o `Limpador de Texto`, que remove *links* e citações que aparecem no texto e o `Stemmer`, que faz a extração do radical de todas as palavras contidas no texto do *tweet*. Tal ferramenta poderia ser utilizada, por exemplo, para o pré-processamento dos dados para que os mesmos possam ser utilizados como entrada de outros algoritmos, como algum método de mineração de dados.

Por meio de uma avaliação empírica, observou-se que alguns filtros da ferramenta descrita possuem maior complexidade de processamento, enquanto outros filtros são pouco exigidos em termos de tempo de processamento útil. Por tal motivo, foi realizada a variação do número de instâncias dos filtros com maior complexidade, mantendo-se inalterado o número de instâncias dos demais filtros da ferramenta. A tabela 3.1 mostra as configurações utilizadas.

Tabela 3.1: Distribuição de instâncias para o processamento de dados do *Twitter*.

Filtro	**Número de Instâncias**
`Extrator de Texto`	2
`Extrator de Hashtags`	2
`Limpador de Texto`	2
`Contador de Palavras`	2
`Contador de Hashtags`	2
`Leitor`	1 – 7
`Removedor de Stopwords`	1 – 7
`Stemmer`	1 – 7

Para cada configuração, o tempo total gasto para o processamento da base de *tweets* foi medido e o *speedup* foi calculado a partir do tempo gasto por uma implementação sequencial da ferramenta. A figura 3.9 (a) mostra o tempo de execução em minutos e a figura 3.9 (b) mostra o *speedup* obtido.

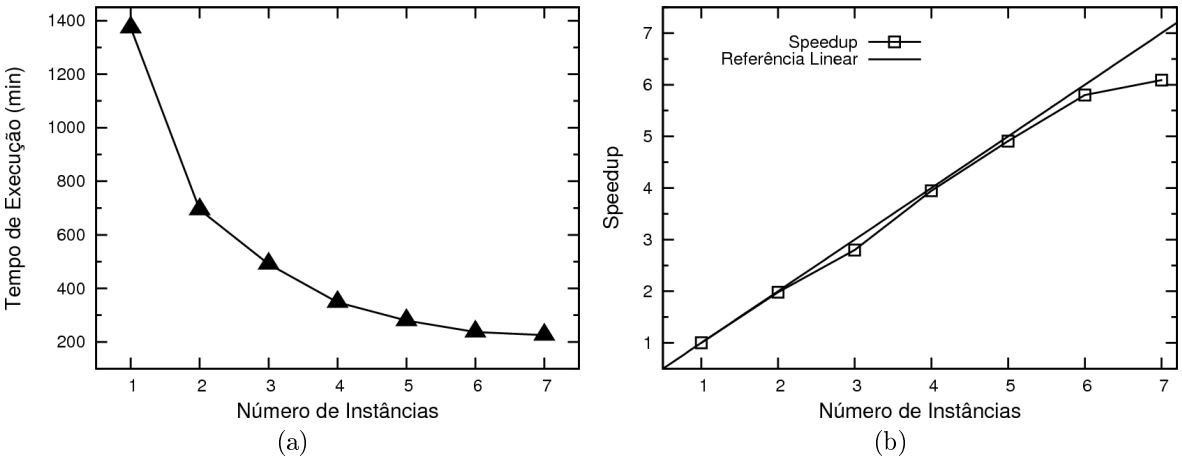


Figura 3.9: Tempo de execução e *speedup* medidos na ferramenta de processamento de dados do *Twitter*.

As curvas obtidas mostram que a ferramenta escala de forma apropriada com a replicação dos principais filtros de processamento. O *speedup* observado é muito próximo

da referência linear, o que significa que o ambiente Watershed não impõe sobrecarga significativa ao processamento como um todo. A partir da replicação com 7 instâncias, é possível verificar uma degradação na cruva de *speedup*, que é explicada pela saturação dos filtros que não foram replicados além de 2 instâncias. Calculando a taxa de processamento atingida nesse experimento, encontramos os valores de aproximadamente 121 transações/s para a configuração com 1 instância de cada filtro principal e 739 transações/s usando 7 instâncias de cada filtro principal, sendo um valor satisfatório para o objetivo da ferramenta.

3.3.2 Paralelização de Algoritmos de Mineração de Dados

Nesta seção são apresentados os resultados da paralelização em Watershed de três algoritmos típicos da área de mineração de dados: Apriori, k-NN e k-Means.

3.3.2.1 Paralelização do Algoritmo Apriori

O algoritmo Apriori [Agrawal et al., 1993] consiste em um método para descoberta de padrões de associação de itens frequentes em grandes conjuntos de dados. A entrada do algoritmo inclui um número ρ, denominado suporte e um conjunto de transações $L = \{t_1, t_2, \ldots, t_n\}$. Cada transação é constituída por um subconjunto dos atributos disponíveis na base de dados, ou seja, $t_i \subset A$, $A = \{a_1, a_2, \ldots, a_m\}$. O objetivo do algoritmo é descobrir quais subconjuntos de atributos ocorrem em mais de ρ transações. O algoritmo utiliza um mecanismo de poda para reduzir significativamente o número de candidatos gerados. Isso é feito com base na propriedade de que nenhum superconjunto de um conjunto de itens infrequente é frequente, o que é utilizado para gerar apenas candidatos potencialmente frequentes.

A implementação em Watershed é composta por três filtros: o `Leitor`, que realiza a leitura das transações da base de dados, o `Contador`, que gera candidatos e o `Agregador`, que combina resultados parciais gerados pelo `Contador`, como pode ser visto na figura 3.10.

Em uma primeira etapa, o `Contador` gera candidatos por meio da contagem de ocorrências de cada atributo presente nas transações recebidas do `Leitor` via política *Round-robin*. Os pares contendo cada candidato e o número de ocorrências são enviados para o `Agregador`, que faz a combinação das contagens parciais. Os atributos infrequentes são eliminados e o resultado final para os conjuntos de itens de tamanho 1 é obtido. Os conjuntos frequentes são informados ao filtro `Contador` para a geração dos candidatos de tamanho 2. O processo é repetido até que todos os conjuntos de itens possíveis sejam gerados. O filtro `Agregador` utiliza a política *Fluxo Rotulado* para

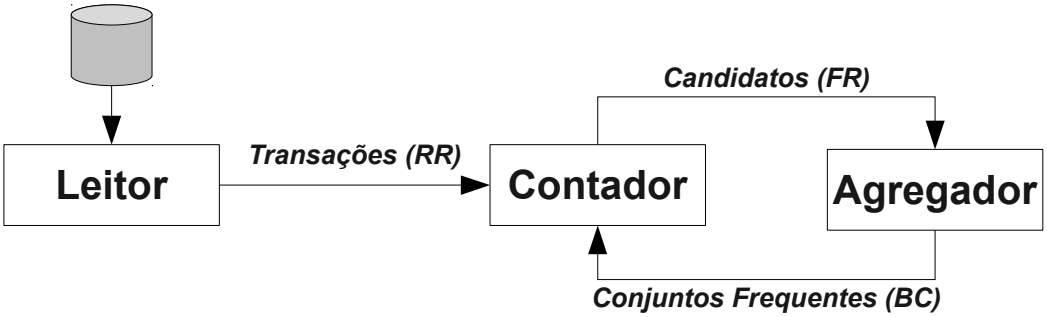
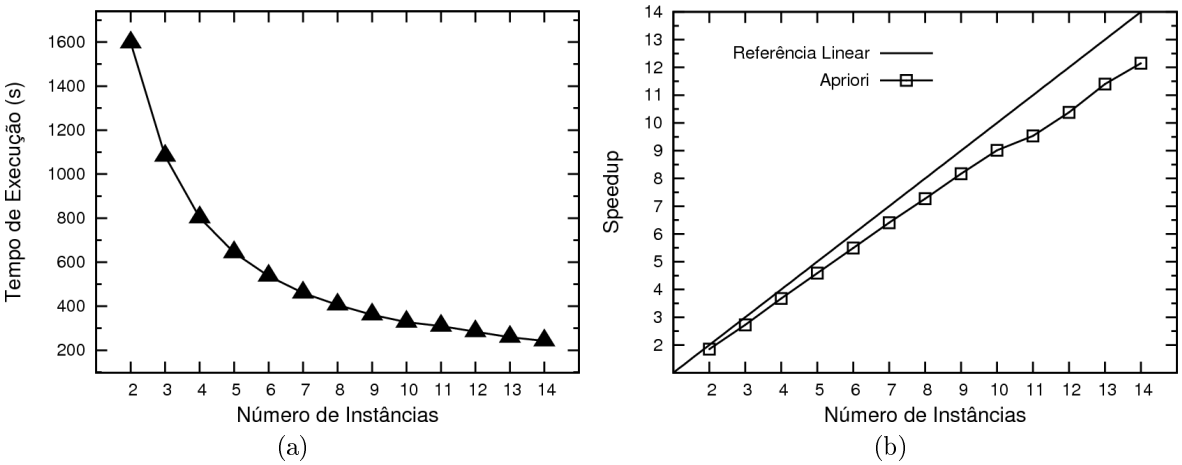


Figura 3.10: Implementação do algoritmo Apriori em Watershed.

distribuir as mensagens entre suas instância, de forma que contagens parciais de determinado candidato sejam entregues sempre à mesma instância. Já o filtro `Contador` consome dados do fluxo de conjuntos frequentes utilizando a política *Broadcast*, uma vez que todas as suas instâncias devem ser informadas sobre quais conjuntos devem ser expandidos em cada iteração do algoritmo.

O experimento realizado utiliza uma base sintética contendo 100 mil transações, cada uma com, no máximo, 20 atributos de um total de 50 disponíveis. O suporte utilizado foi 2% da base de dados. O tempo de execução da versão sequencial do mesmo algoritmo é de aproximadamente 2.949 segundos. Nesse experimento, o número de instâncias do filtro `Agregador` foi mantido em 4 para todas as execuções, uma vez que o mesmo não realiza computações intensivas. Os demais filtros receberam de 2 a 14 instâncias cada, tendo em vista a demanda computacional dos mesmos.



(a) (b)

Figura 3.11: Tempo de execução e *speedup* medidos na execução do algoritmo Apriori.

Os gráficos da figura 3.11 apresentam os resultados obtidos para a paralelização do algoritmo Apriori. Como pode ser visto, a aplicação escala quase linearmente com

o aumento do número de instâncias dos filtros críticos.

3.3.2.2 Paralelização do Algoritmo k-NN

O algoritmo k-NN, *K Nearest Neighbors* [Witten & Frank, 2002], é uma técnica de associação que classifica elementos de um conjunto S de transações baseado nos k elementos mais semelhantes obtidos de uma base de treino T. O algoritmo k-NN é um exemplo de aplicação embaraçosamente paralela, ou seja, a computação pode ser facilmente dividida em partes independentes, que podem ser executadas em elementos de processamento distintos. Além disso, trata-se de uma aplicação típica de processamento contínuo, onde cada transação é classificada sem dependência das demais transações da base de teste. A implementação em Watershed, ilustrada na figura 3.12, conta com 4 filtros de processamento.

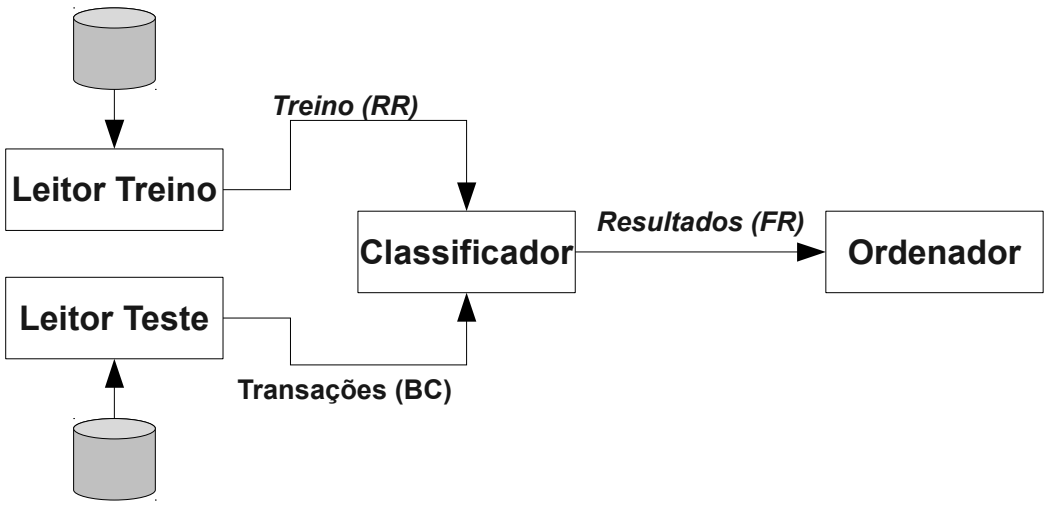


Figura 3.12: Implementação do algoritmo k-NN em Watershed.

O filtro `Leitor Treino` faz a leitura da base de treino e escreve no fluxo lido pelo filtro `Classificador` utilizando a política *Round-robin*, de maneira que a base de treino fica dividida entre suas instâncias. O filtro `Classificador` consome as transações produzidas pelo filtro `Leitor Teste` utilizando a política *Broadcast*. Após receberem suas porções da base de treino, as instâncias do filtro `Classificador` realizam uma classificação local das transações recebidas utilizando a distância Euclidiana como métrica de similaridade. O resultado local é então escrito em um fluxo que é lido pelo último filtro da cadeia de processamento, o `Ordenador`, utilizando a política de *Fluxo Rotulado*. Esse filtro é responsável por ordenar os resultados parciais obtidos pelo filtro anterior, gerando como saída os k itens mais similares à transação sendo classificada.

Nesse experimento foi utilizada uma base de treino com mil itens e um conjunto de teste com 100 mil transações em um espaço vetorial de 20 dimensões, sendo as duas bases geradas sinteticamente. Para o parâmetro k foi utilizado o valor 50. O tempo de execução da versão sequencial do algoritmo para processar os dados com a mesma configuração foi de aproximadamente 2.328 segundos. Os filtros foram configurados da seguinte maneira: 1 única instância do `Leitor Treino`, uma vez que esse filtro não demanda muito processamento, 2 instâncias do filtro `Ordenador` e uma variação de 2 a 14 instâncias dos filtros `Leitor Teste` e `Classificador`, que são os filtros de maior demanda computacional.

Os gráficos da figura 3.13 mostram os resultados obtidos nos experimentos com o algoritmo k-NN.

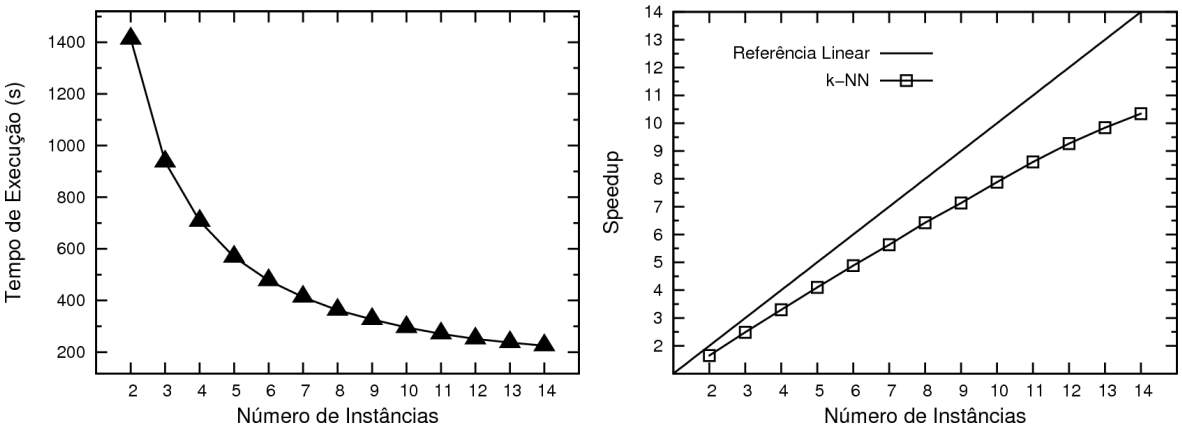


Figura 3.13: Tempo de execução e *speedup* medidos na execução do algoritmo k-NN.

O gráfico de *speedup* mostra que o algoritmo mostrou-se escalável, embora tenha sido observada uma diminuição do ganho de desempenho com o aumento da replicação dos filtros de processamento. Esse comportamento é causado pelo aumento significativo nos dados trafegados no fluxo de transações, uma vez que as mesmas são lidas pelo filtro `Classificador` utilizando-se a política de distribuição *Broadcast*.

3.3.2.3 Paralelização do Algoritmo k-Means

O algoritmo k-Means [Macqueen, 1967] é um dos algoritmos mais simples para classificação não-supervisionada e mais amplamente utilizado, apresentando bons resultados para o problema de agrupamento de dados. A ideia central do k-means consiste em formar k grupos de dados, utilizando a estratégia de maximizar a distância intergrupos à medida que minimiza a distância intragrupo. A entrada do algoritmo é composta

por um conjunto de transações a serem agrupadas e o número k de grupos a serem gerados. O algoritmo funciona de forma iterativa, definindo elementos centrais, denominados centróides, para cada grupo com base em alguma distância de similaridade entre as transações a serem agrupadas.

A figura 3.14 apresenta a paralelização do algoritmo em Watershed. Nessa implementação foram utilizados 3 filtros de processamento. O filtro `Leitor` é responsável por obter as transações da base de dados e escrevê-las em um fluxo a ser lido pelo próximo filtro da cadeia de processamento. O filtro `Classificador` consome as transações produzidas, distribuindo-as alternadamente entre suas instâncias. Cada instância classificadora realiza o agrupamento do subconjunto de transações ao qual tem acesso e escreve os centróides calculados localmente em um fluxo de saída. O último filtro da aplicação, denominado `Concentrador`, lê os centróides locais, combinando-os em centróides globais, que são informados a todas as instâncias do filtro `Classificador` para a próxima iteração. O processo é repetido até que não haja mudanças significativas nos centróides globais ou até que um número máximo de iterações seja alcançado.

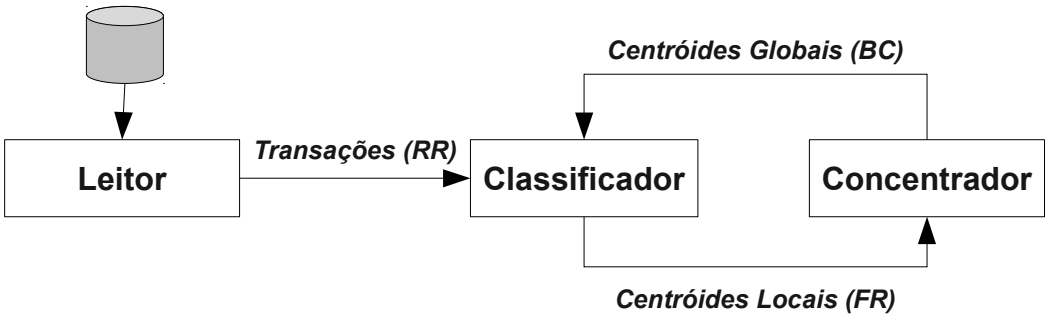
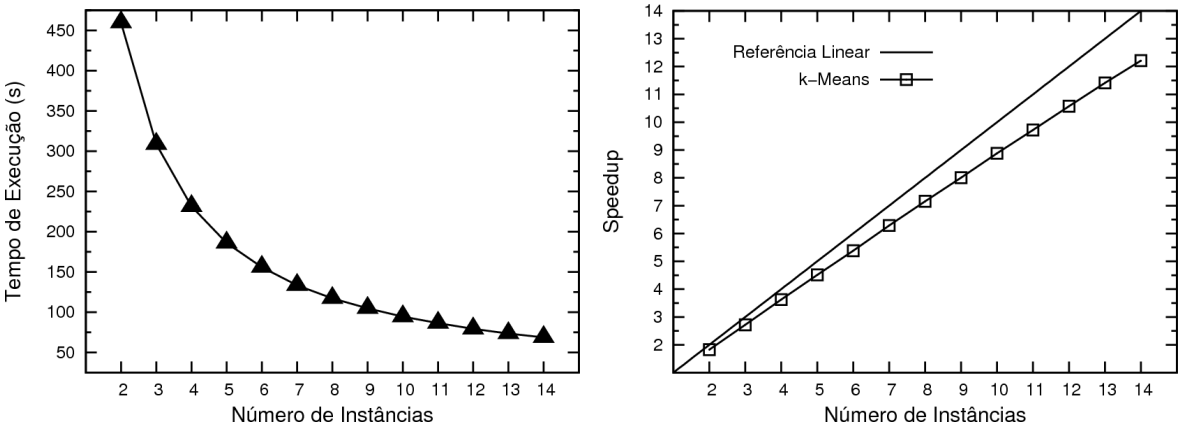


Figura 3.14: Implementação do algoritmo k-Means em Watershed.

O experimento foi realizado com uma base de dados contendo 100 mil transações em um espaço de 20 dimensões. Utilizou-se $k = 200$ como o número de grupos a serem formados em no máximo 50 iterações. O tempo de execução da versão sequencial do algoritmo é de aproximadamente 839 segundos. Estruturalmente, as instâncias dos filtros `Leitor` e `Classificador` foram variadas de 2 a 14, mantendo-se o filtro `Concentrador` com 4 instâncias. Os resultados para o tempo de execução e respectivo *speedup* são mostrados na figura 3.15.

Pela figura, é possível observar que o algoritmo k-means apresenta *speedup* próximo da referência linear, sendo que a curva obtida é a que contém menor variação no ganho de desempenho dentre os três algoritmos de mineração de dados paralelizados. Isso ocorre porque o algoritmo k-means é o mais estável em termos do número de men-



(a) (b)

Figura 3.15: Tempo de execução e *speedup* medidos na execução do algoritmo k-Means.

sagens trocadas entre as instâncias de seus filtros, não sofrendo variações significativas com o aumento do nível de paralelismo de dados realizado.

3.4 Considerações Finais

Nesse capítulo foi apresentado o ambiente de processamento distribuído de fluxos de dados denominado Watershed. Foram descritas as funcionalidades básicas, o modelo de programação utilizado e a arquitetura em camadas do ambiente.

Watershed é caracterizado pela dinamicidade e flexibilidade na construção de aplicações distribuídas. Ele oferece abstrações de alto nível que possibilitam a exploração de ambos os níveis de paralelismo, o de tarefas e o de dados, permitindo o processamento *online* e *offline* de grandes volumes de dados de forma eficiente. Adicionalmente, o ambiente permite alterações topológicas nas aplicações em tempo de execução. Os experimentos realizados mostram que o *overhead* imposto por Watershed às aplicações utilizadas não alcança níveis significativos, sendo possível observar escalabilidade em todos os casos estudados.

O próximo capítulo trata das mudanças estruturais nos filtros de aplicação. Essas mudanças incluem a alteração no número de instâncias em execução e a migração de instâncias entre máquinas de processamento.

Capítulo 4

Reconfiguração Dinâmica em Aplicações Watershed

Uma reconfiguração dinâmica tem como principal objetivo permitir que uma sistema evolua de sua configuração atual C_i para uma nova configuração C_{i+1}, geralmente em resposta a mudanças ocorridas no ambiente de execução. Uma configuração pode ser definida como o conjunto de entidades de *software* presentes na aplicação em execução em um dado instante. Essas entidades podem ser estruturas de dados, canais de comunicação, mensagens e outros. No caso ideal, uma reconfiguração dinâmica não deve impor sobrecarga à execução do sistema. Contudo, isso é praticamente impossível, sendo que, na prática, busca-se produzir o menor impacto possível.

Este capítulo descreve o mecanismo de reconfiguração dinâmica projetado para aplicações de processamento distribuído de fluxos de dados. A estratégia utilizada assume o modelo de programação e as características de funcionamento do ambiente Watershed, descritos no capítulo 3.

O capítulo está organizado da seguinte maneira: a seção 4.1 relaciona os principais requisitos e desafios envolvidos na tarefa de reconfiguração dinâmica. Na seção 4.2 é descrita a estratégia proposta para viabilizar a reconfiguração dinâmica e a arquitetura do mecanismo. Na seção 4.3 são detalhados alguns problemas originados dos requisitos estabelecidos e as respectivas soluções adotadas no mecanismo proposto. Por fim, a seção 4.4 discute alguns detalhes de implementação e utilização.

4.1 Requisitos Fundamentais

O projeto de um ambiente de computação distribuída envolve, entre outros aspectos, uma série de requisitos fundamentais para o bom funcionamento e boa usabilidade.

Um desses requisitos é a provisão de mecanismos transparentes para que o programador de aplicação atenha-se ao domínio específico do problema que deve resolver. A propriedade de transparência deve ser parte considerada em qualquer mecanismo integrado ao ambiente, ou seja, cada funcionalidade adicionada deve minimizar o esforço do programador em sua utilização. Dessa forma, uma estratégia de reconfiguração dinâmica deve exportar, por meio de uma interface simples, suas funcionalidades e, da forma mais transparente possível, executar todas as ações necessárias para que os objetivos sejam alcançados.

Além do requisito de transparência, dois outros requisitos fundamentais relacionados a ambientes distribuídos de processamento de larga escala merecem destaque. O primeiro requisito diz respeito à composição e escalabilidade das aplicações, que é derivado do fato de que a maioria das aplicações geralmente são construídas por meio da composição entre elas próprias e outras aplicações já existentes. Com base nesse requisito, o projeto do ambiente deve permitir que várias aplicações possam interagir entre si, por meio do compartilhamento de dados. Contudo, a propriedade de escalabilidade deve ser aplicada de forma justa, sem que o crescimento de uma aplicação influencie negativamente no desempenho das demais aplicações. Em particular, uma reconfiguração dinâmica não deve causar impactos significativos em aplicações não participantes da ação.

O segundo requisito diz respeito à dependência entre processos, que é derivado da observação de que as aplicações podem ser muito complexas em estrutura, contendo muitas dependências temporais e de dados entre os estágios de processamento. Dessa forma, os estágios de processamento das aplicações devem ser escalonados de tal maneira que respeitem essas dependências. Esse requisito deve ser tratado com mais ênfase nos mecanismos de escalonamento de tarefas ou balanceamento de carga, que não são contemplados no presente trabalho. Com relação ao mecanismo de reconfiguração dinâmica, o requisito é levado em consideração para a manutenção estrutural e semântica das aplicações em execução no ambiente Watershed.

Relacionados alguns dos requisitos fundamentais típicos de ambientes de computação distribuída, faz-se necessária a definição dos principais requisitos intimamente relacionados à tarefa de reconfiguração dinâmica. A estratégia proposta no presente trabalho apóia-se sobre os seguintes requisitos:

- *Correção.* Deve haver mecanismos para a obtenção de uma evolução correta, tanto das aplicações executando quanto do ambiente de execução. Em outras palavras, os resultados gerados pelas aplicações devem ser os mesmos ocorrendo ou não ações de reconfiguração.

- *Overhead.* O impacto de uma reconfiguração deve ser minimizado e a presença do mecanismo por si não deve representar uma sobrecarga significativa durante a operação normal do ambiente.

- *Aplicabilidade.* A estratégia deve ser aplicável a qualquer aplicação dentro do modelo de computação utilizado pelo ambiente de execução. Esse requisito inclui aplicações simples ou complexas, sem estado e com estado, como será descrito em breve.

- *Escalabilidade.* A estratégia deve ser escalável, uma vez que o propósito do ambiente de computação distribuída inclui essa característica. Além disso, os mecanismos utilizados devem favorecer a escalabilidade das aplicações em execução, não sendo um entrave ao aumento do desempenho dessas.

- *Transparência.* A reconfiguração precisa ser feita da maneira mais transparente possível, requerendo o mínimo de conhecimento e esforço dos desenvolvedores de aplicação.

- *Assincronia.* O modelo de computação utilizado no ambiente Watershed permite a exploração do paralelismo de dados e do paralelismo de tarefas. A combinação desses tipos de paralelismo possibilitam que o processamento seja realizado, em sua maioria, de forma assíncrona. A propriedade de assincronia deve ser mantida no maior nível possível, para que o desempenho das aplicações não seja afetado na ocorrência de reconfigurações.

O processo de reconfiguração de um sistema em execução é, por definição, um processo intrusivo. Isso porque uma reconfiguração dinâmica pode envolver a criação, remoção, migração e mudanças nos relacionamentos entre entidades ditas reconfiguráveis. No caso particular do ambiente Watershed, uma reconfiguração pode ser feita por meio da adição, remoção ou migração de instâncias de filtros de processamento. Independente da operação realizada, a interação entre o filtro reconfigurado e seus filtros produtores e consumidores é, consequentemente, modificada. Devido a essa característica intrusiva, o mecanismo de reconfiguração precisa assegurar que as entidades componentes do sistema e suas interações funcionem corretamente durante e após a transição entre configurações, como será detalhado adiante.

4.2 Mecanismo de Reconfiguração Dinâmica

O mecanismo de apoio a reconfiguração proposto funciona, basicamente, por meio de operações realizadas sobre os filtros de aplicações Watershed. Essas operações promovem a elasticidade dessas entidades e a modificação dos recursos a elas alocados. As operações admitidas incluem: *adição de instâncias*, *remoção de instâncias* e *migração de instâncias*. Com as duas primeiras operações é possível acomodar mudanças na carga de trabalho de um filtro, aumentando ou diminuindo o número de instâncias de acordo com a demanda por processamento. A operação de migração permite que as aplicações sejam reestruturadas, de maneira que as instâncias de seus filtros possam mudar de servidores. Todas essas operações são realizadas com o intuito de aumentar o desempenho das aplicações em execução e fazer uma melhor utilização dos recursos computacionais presentes no *cluster* de processamento.

4.2.1 Arquitetura

A figura 4.1 ilustra a arquitetura geral do mecanismo proposto para reconfiguração dinâmica de aplicações executando no ambiente Watershed. O mecanismo foi implementado como um módulo dedicado do Gerente de Execução, que é parte da camada de controle de Watershed, como descrito na seção 3.2.2.

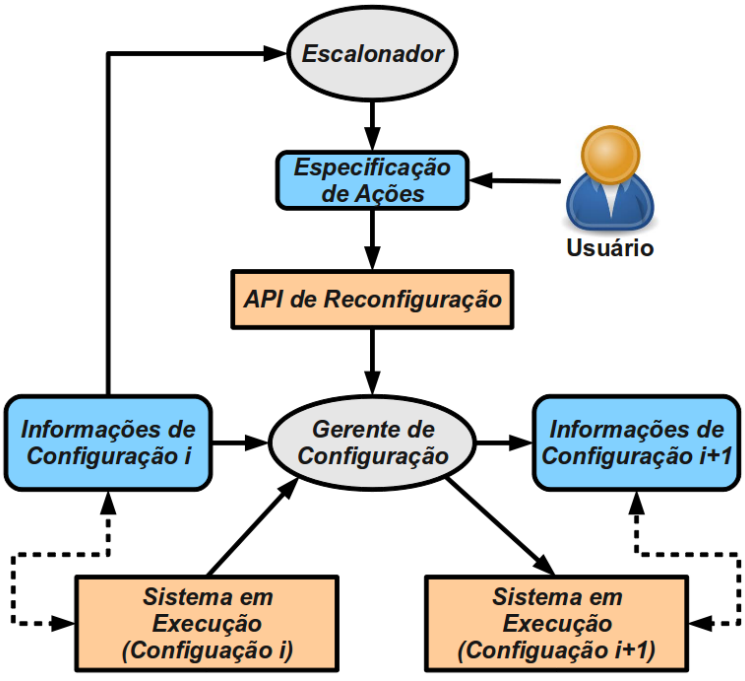


Figura 4.1: Arquitetura do mecanismo de reconfiguração.

Nessa arquitetura, o componente denominado *Escalonador* recebe periodicamente, do ambiente, um conjunto de informações sobre as aplicações em execução, como o tempo médio de processamento de mensagens por filtro, a carga de trabalho nos servidores de processamento, entre outros. Com base nessas informações e em um modelo de desempenho, o *Escalonador* é capaz de produzir especificações precisas de ações a serem tomadas durante a execução do sistema. Cada especificação pode ser vista como um roteiro que leva o sistema de uma configuração C_i a uma configuração C_{i+1}. As ações que compõem uma especificação são relacionadas em termos das entidades reconfiguráveis e de operações a serem realizadas sobre essas entidades.

A tabela 4.1 mostra um exemplo de especificação de reconfiguração. Como pode ser visto, a transição entre configurações pode envolver várias entidades e várias ações. Nesse exemplo, o *Escalonador* especifica que deve-se migrar a instância 3 do filtro A para o servidor $S4$, remover a instância 4 do filtro A e adicionar uma instância do filtro B no servidor $S3$. É importante observar que as ações não precisam especificar o servidor de origem seja qual for a operação, pois o gerente de configuração possui informações sobre cada uma das instâncias em execução.

Tabela 4.1: Exemplo de especificação de reconfiguração.

Filtro	**Instância**	**Ação**	**Destino**
A	3	Migrar	S4
A	4	Remover	–
B	–	Adicionar	S3

Além da utilização do escalonador automático, o mecanismo proposto oferece a possibilidade de interação com o usuário do sistema, que pode, por meio do *console* de comandos do ambiente (seção 3.2.2), especificar ações de reconfiguração. Nesse último caso, o usuário deve desabilitar o escalonador para que ele não desfaça suas decisões no futuro. Os comandos relacionados às operações de reconfiguração disponíveis no *console* do ambiente são:

- `add-instance [filter_name] [destination_host]`: comando para adicionar uma instância a algum filtro em execução. Recebe como argumentos o nome do filtro e o servidor de processamento onde a instância deverá ser carregada.

  Exemplo de utilização: `watershed add-instance B S3`

- `remove-instance [filter_name] [instance_rank]`: comando para remover uma instância ativa. Recebe como argumento o nome do filtro e o identificador da instância a ser removida.

Exemplo de utilização: `watershed remove-instance A 4`

- `migrate-instance [filter_name] [instance_rank] [destination_host]`: comando utilizado para realizar a migração de uma instância em execução entre máquinas do *cluster*. Recebe como argumentos o nome do filtro, o identificador da instância e o nome da máquina de destino.

  Exemplo de utilização: `watershed migrate-instance A 3 S4`

A especificação de reconfiguração é então interpretada pela *API de Reconfiguração*, que exporta as funções disponíveis para a execução das ações. As ações são aplicadas sob a supervisão do *Gerente de Configuração*, que é responsável por fazer com que o sistema seja modificado, de maneira correta e consistente, passando de sua configuração atual para uma nova configuração. O *Gerente de Configuração* utiliza as informações de configuração atual para que sejam mantidas as restrições de integridade e correção detalhadas na seção 4.3.

4.2.2 Operações de Reconfiguração

ADIÇÃO

A operação de adição de instâncias permite que um filtro de aplicação seja expandido em tempo de execução. Em outras palavras, novas cópias de processamento de um filtro são adicionadas ao ambiente, aumentando assim o poder de processamento do filtro em questão. A figura 4.2 ilustra a adição de uma instância ao filtro de processamento R.

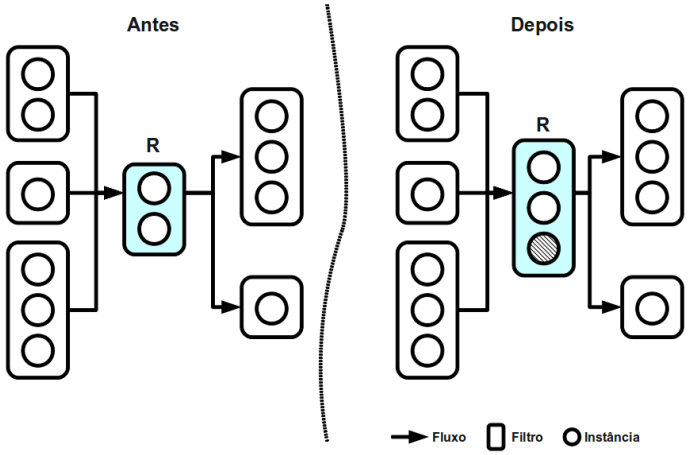


Figura 4.2: Operação de adição de instância de processamento.

Como pode ser visto na figura, a adição de uma instância não altera as relações existentes entre o filtro sendo reconfigurado, seus produtores e seus consumidores. Os dados continuam sendo produzidos e consumidos utilizando-se as políticas de distribuição de mensagens definidas na configuração inicial dos filtros envolvidos.

A partir de sua adição, a nova instância assume parte da computação do filtro, recebendo uma porção dos dados já consolidados nas demais instâncias. Além disso, a nova instância passa a receber parte das mensagens geradas a partir do momento de sua adição, conforme definido para cada fluxo de entrada do filtro R.

Remoção

A operação de remoção permite que um filtro de aplicação tenha o número de instâncias diminuído. Essa operação é útil no caso em que as instâncias do filtro encontram-se ociosas, sendo interessante diminuir o número de processos em execução, liberando recursos para outros filtros. A operação de remoção é exemplificada na figura 4.3.

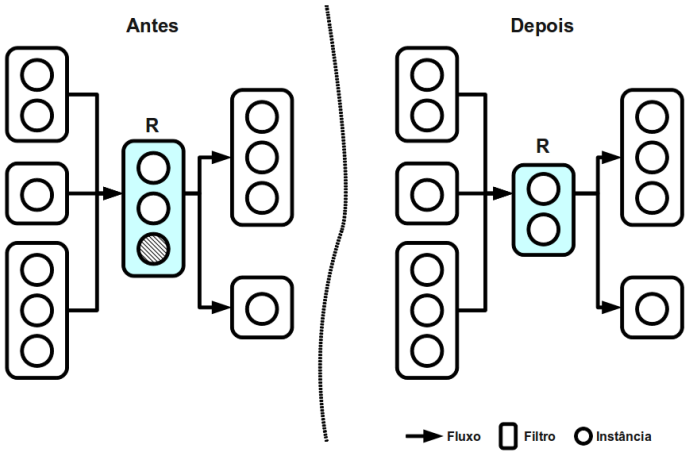


Figura 4.3: Operação de remoção de instância de processamento.

Após a remoção de uma instância, os dados consolidados por ela são distribuídos entre as instâncias que permanecem ativas. O mesmo acontece com espaço de mensagens processado pela instância removida, ou seja, todas as mensagens antes direcionadas à instância sendo removida passam a ser direcionadas às instâncias que permanecem no ambiente.

Migração

A migração consiste em mover uma instância de um servidor de processamento para outro. Nessa operação, a instância preserva sua identidade e estado, não

sendo necessária a etapa de redistribuição de dados consolidados e mensagens futuras. Os dados do estado da instância sendo migrada são salvos em meio persistente e enviados para o processo criado no servidor de destino, que reinicia o processamento.

A migração, assim como as demais operações, é realizada de maneira transparente ao programador de aplicação. Com isso, não é necessária nenhuma ação da aplicação para promoção das operações de reconfiguração.

A figura 4.4 ilustra a migração de uma instância de seu local original, servidor x para o novo local de execução, servidor y.

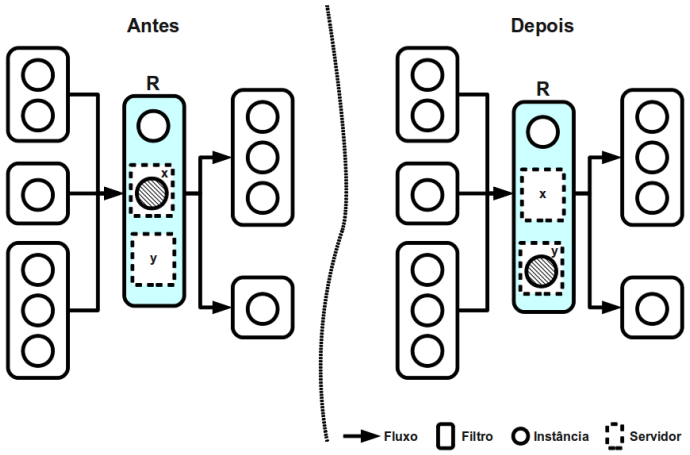


Figura 4.4: Operação de migração de instância de processamento.

PROTOCOLO DE FUNCIONAMENTO

Como descrito anteriormente, as instâncias de um filtro de aplicação são processos dispostos em um grupo MPI dedicado. Essa estratégia facilita o controle de comunicação entre processos de um mesmo filtro e entre processos de filtros distintos. Cada fluxo de dados é implementado como sendo um canal de comunicação entre um grupo de processos produtores e um grupo de processos consumidores. Por essa razão, as operações de reconfiguração são realizadas por meio de um conjunto de manipulações sobre os processos e canais de comunicação, seguindo um protocolo bem definido para tal.

O protocolo geral de funcionamento de uma reconfiguração é, em sua grande parte, comum às três operações disponíveis. Abaixo são descritos, de forma simplificada, os principais passos do protocolo.

1. O *Escalonador* ou o usuário do ambiente faz uma requisição de reconfiguração, fornecendo a especificação da operação.

2. A *API de Reconfiguração* faz a interpretação da operação e repassa ao *Gerente de Configuração*.
3. O *Gerente de Configuração* envia uma mensagem a todas as instâncias do filtro R a ser modificado e a todas as instâncias de seus produtores e consumidores. Essa mensagem contém a indicação de reconfiguração do filtro R.
4. As instâncias produtoras suspendem o envio de dados ao filtro R até que a reconfiguração seja completada. A suspensão do envio é comunicada pelos filtros produtores ao filtro R, utilizando uma etapa de sincronização.
5. As instâncias de R processam as mensagens em trânsito, enviadas antes da sincronização realizada. Após a sincronização com os produtores, o filtro R realiza uma sincronização com seus consumidores para garantir que esses tenham recebido todas as mensagens em trânsito partindo de R.
6. O filtro R comunica ao *Gerente de Configuração* que está pronto para ser alterado.
7. Se a operação é Adicionar
   a) O *Gerente de Configuração* dispara um novo processo n do filtro R.
   b) O processo n se conecta ao grupo de instâncias de R. As conexões entre os filtros são refeitas para que n faça parte das mesmas.
8. Se a operação é Remover
   a) O *Gerente de Configuração* envia mensagem ao filtro R, informando qual é a instância que deve ser removida.
   b) Os dados do estado da instância sendo removida são armazenados em memória secundária, caracterizando uma operação de *checkpoint*.
   c) A instância a ser removida é desconectada do grupo de processos de R. Além disso, ela é removida das conexões com os produtores e consumidores de R.
9. Se a operação é Migrar
   a) O passo 8 é executado, seguido do passo 7.
10. O *Gerente de Configuração* redistribui os blocos de trabalho[1] entre as instâncias de R, replicando as informações da nova configuração entre produtores e consumidores de R. No caso de uma migração, a distribuição dos blocos de trabalho continua inalterada.

[1] Os blocos de trabalho são entidades que definem a atuação de cada processo sobre o domínio do problema e conjunto de mensagens – esse conceito será detalhado na seção 4.3

11. Os produtores, consumidores e o próprio R são notificados pelo *Gerente de Configuração* sobre a conclusão da reconfiguração. A partir desse momento, os dados voltam a ser entregues às instâncias de R.

O mecanismo de reconfiguração proposto é estruturado de maneira simples, viabilizando a execução das três operações básicas descritas. Com essas operações, é possível modificar as aplicações em execução, em resposta a mudanças ocorridas no ambiente ou na carga de trabalho imposta aos filtros de processamento. Alguns problemas relacionados à execução dessas operações incluem a manutenção da consistência e correção das aplicações em execução no ambiente entre configurações diferentes. Esses problemas e suas respectivas soluções são explorados na próxima seção.

4.3 Manutenção de Consistência

Um dos requisitos fundamentais para o bom funcionamento do mecanismo de reconfiguração é a manutenção da consistência das aplicações. Por consistência, define-se que o resultado da computação de uma aplicação que tenha passado por reconfigurações deve ser exatamente igual ao resultado da mesma aplicação sem a presença de reconfigurações. No caso de aplicações sem estado, ou seja, aplicações que não necessitam armazenar dados de importância semântica, as reconfigurações não influenciam na consistência de seus resultados. Por outro lado, em aplicações que necessitam armazenar dados gerados pelo processamento de mensagens que interferem no processamento das mensagens subsequentes, uma operação de reconfiguração pode gerar resultados incorretos, caso não sejam adotadas estratégias que previnam essas anomalias. Esta seção tem como objetivo elucidar o problema de consistência e apresentar as estratégias utilizadas para contornar tal problema.

4.3.1 Aplicações com Estado Particionado

O particionamento do domínio das mensagens em aplicações Watershed é realizado, a exemplo do que ocorre no ambiente Anthill, por meio da utilização da política de entrega de mensagens denominada fluxo rotulado. Nessa política, o desenvolvedor de um determinado filtro implementa uma função de rotulação, que é utilizada para distribuir as mensagens de entrada entre as instâncias do filtro. Tipicamente, o rótulo de uma mensagem é submetido a uma operação modular, utilizando o número de instâncias do filtro receptor, de forma que a mensagem possa ser entregue a alguma instância válida:

$$d = |\text{ROTULO}(m)| \; mod \; |I_F| \qquad \text{ROTULO}(m) \in \mathbb{Z} \tag{4.1}$$

onde d é a instância do conjunto I de instâncias do filtro F, escolhida como destino da mensagem m, rotulada pela função ROTULO.

Na prática, a aplicação dessa função gera o particionamento do estado do filtro, uma vez que cada instância será responsável por processar mensagens de certa forma similares, de acordo com a função de rotulação. Como exemplo, seja um filtro F cuja operação seja contar palavras. Supondo que esse filtro contenha n instâncias e que a função de rotulação seja tal que:

$$\text{ROTULO}(m) = \sum_{i=1}^{|w|} w[i]$$

onde w é a palavra que foi extraída da mensagem m e $w[i]$ é a representação numérica do i-ésimo caractere de w.

Cada uma das n instâncias será responsável por contar um determinado conjunto de palavras e não haverá nenhuma palavra sendo contada por mais de uma instância. Como exemplo, a situação hipotética abaixo poderia ser constatada.

$$\begin{aligned}
\text{SAIDA}(F_1) &= \{caneta^2, computador^3, boracha^1\} \\
\text{SAIDA}(F_2) &= \{mochila^5, apagador^1\} \\
&\vdots \\
\text{SAIDA}(F_n) &= \{mesa^{15}, cadeira^{10}, caderno^{16}\}
\end{aligned}$$

A aplicação da função tradicional de *hash* sobre o número de instâncias de um filtro é muito simples e funciona bem na ausência de reconfigurações. Todavia, apresenta limitações graves quando uma reconfiguração se faz necessária.

Seja o mesmo exemplo de filtro contador de palavras. Caso uma operação de migração seja aplicada a alguma instância, nenhum problema é observado. Por outro lado, caso uma instância seja acrescentada ou removida do filtro F, a distribuição das mensagens será comprometida. No caso da inserção de uma instância, o mapeamento das mensagens seria dado por $d = \text{ROTULO}(m) \; mod \; (n+1)$ e para a remoção de uma instância seria dada por $d = \text{ROTULO}(m) \; mod \; (n-1)$.

O resultado é que grande parte das mensagens, $\frac{n}{n+1}$ ou $\frac{n}{n-1}$, em média, geradas depois da reconfiguração, devem ser redistribuídas de forma desorganizada entre as instâncias do filtro F. Considerando-se apenas a redistribuição das mensagens, o

problema seria constatado no resultado da computação. Esse problema é denominado *hash disruption*. Para o exemplo do contador de palavras, mais de uma instância teria a contagem de uma palavra em comum como resultado, pois o mapeamento das palavras para as instâncias do filtro antes e depois da reconfiguração seria diferente.

Uma solução para esse problema seria mover os dados armazenados em cada instância para os novos responsáveis pelo processamento, utilizando o mesmo tratamento dado às mensagens geradas após uma reconfiguração. Essa estratégia, embora pareça funcionar, contém dois problemas graves. O primeiro diz respeito à movimentação de dados, pois grande parte dos dados já processados, que fazem parte do estado particionado entre as instâncias, deve ser movimentada para atender o novo mapeamento. Isso pode tornar o mecanismo de reconfiguração ineficiente a ponto de se tornar inviável. O outro problema é a dificuldade de reparticionar os dados já processados. Cada instância deveria ser capaz de dividir os dados que armazena relacionando-os às mensagens processadas, o que de fato, é muito difícil. Além disso essa tarefa ficaria a cargo do programador do filtro, uma vez que os dados armazenados são definidos por ele. Por ir contra os requisitos de eficiência e transparência, essa estratégia não é empregada nesse trabalho.

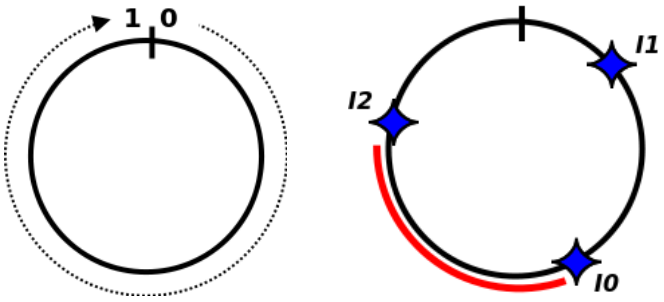
4.3.2 Redistribuição de Mensagens

Uma solução para o problema descrito anteriormente tem origens na área de protocolos para caches distribuídos de aplicações para a Web. A estratégia, denominada *Hash* Consistente [Karger et al., 1997], foi proposta para reduzir o impacto causado pelo problema observado na distribuição de chaves em sistemas distribuídos que utilizam funções *hash* como ferramenta de assinalamento. Na prática, a estratégia foi originalmente utilizada para localizar documentos da *Web* em um conjunto de *caches* variável.

Da mesma maneira que é feita em uma estratégia comum de *hashing*, o *hash* consistente realiza o espalhamento dos dados de um dicionário distribuído entre as máquinas que compõem um *cluster* de processamento. No caso particular de Watershed, as mensagens enviadas por um produtor continuam sendo distribuídas entre as instâncias de um receptor caso a política escolhida seja a de fluxo rotulado. No entanto, caso uma reconfiguração seja realizada, apenas uma porção dos dados constituintes do estado compartilhado precisa ser movida entre instâncias de filtros para acomodar a nova configuração.

A estratégia pode ser ilustrada por uma circunferência de perímetro unitário, que define o espaço de mapeamento de chaves. Sobre cada chave gerada como identificador

de uma mensagem, é aplicada uma função de mapeamento que resulta em um valor no intervalo $[0, 1)$. A figura 4.5 (a) ilustra o mecanismo sem a presença de instâncias de processamento. Nessa figura, a circunferência construída com linha pontilhada indica o sentido de assinalamento de chaves às instâncias de processamento.



(a) (b)

Figura 4.5: Mapeamento de chaves utilizando a estratégia de *hash* consistente.

Por outro lado, a figura 4.5 (b) ilustra o mecanismo na presença de três instâncias de processamento de um determinado filtro. Essas instâncias são simbolicamente dispostas na mesma circunferência de mapeamento de mensagens. Para mapear as instâncias, poderia ter sido utilizada a mesma função aplicada às mensagens. Contudo, esse trabalho utiliza uma estratégia gulosa para o mapeamento de instâncias, como será detalhado mais adiante.

Quando uma mensagem de dados é enviada para um filtro consumidor, o ambiente Watershed faz a entrega dessa mensagem seguindo a política de recebimento estabelecida pelo filtro em questão. Caso a política de recebimento seja de fluxo rotulado, o usuário provê uma função de rotulação f, que recebe como argumento o conteúdo da mensagem m, produzindo uma chave numérica k como resultado:

$$f(m) = k \qquad k \in [0, R) \tag{4.2}$$

A chave k é então utilizada para mapear a mensagem em algum ponto da circunferência. Uma vez que o domínio das chaves geradas é especificado pelo ambiente por meio da constante R, o mapeamento é feito de maneira direta, utilizando-se a seguinte função:

$$h(k) = k \div R \tag{4.3}$$

O mapeamento de m define qual instância será responsável por seu processamento, que será a primeira instância alocada no sentido horário, partindo do ponto de

mapeamento de m, sobre a circunferência de mapeamento de chaves. Como exemplo, uma mensagem cujo mapeamento seja feito para o intervalo destacado na figura 4.5 (b) será entregue à instância I_2.

Em uma situação de reconfiguração, o mecanismo pode aplicar uma das três operações básicas disponíveis. A redistribuição de mensagens a partir do momento da reconfiguração depende da operação realizada. No caso de uma operação de migração, o assinalamento de mensagens às instâncias de processamento não se altera, uma vez que há apenas uma mudança física da instância sendo migrada de um servidor para outro. A operação de remoção, por sua vez, ocasiona uma redistribuição das mensagens antes assinaladas à instância removida. Tais mensagens passam a ser assinaladas à primeira instância alocada na circunferência de mapeamento, seguindo o sentido horário a partir do ponto de remoção, como ilustrado na figura 4.6.

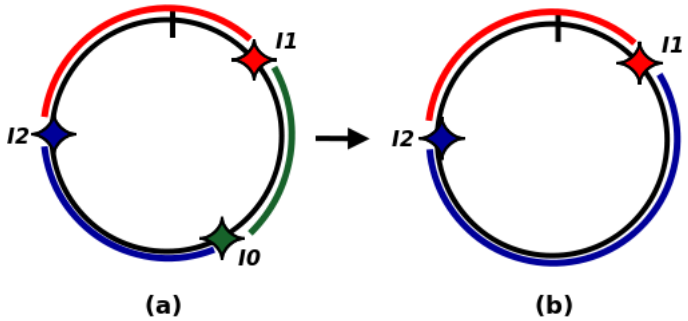


Figura 4.6: Redistribuição de mensagens após uma operação de remoção de instância.

No exemplo da figura, a instância I_0 foi removida e todas as mensagens que seriam direcionadas a ela, serão entregues à instância I_2 após a reconfiguração. O problema dessa abordagem simplificada é o fato de que ao se remover uma instância, o mecanismo pode gerar um desbalanceamento da carga total distribuída entre as instâncias do filtro. Caso fosse realizado o balanceamento de carga no momento da remoção, a propriedade do *hash* consistente seria inutilizada e seria necessária uma grande movimentação de dados entre instâncias de processamento. Como o tempo entre reconfigurações pode ser pequeno, a redistribuição balanceada de partições de estado, que será vista adiante, tornar-se-ia um gargalo para o desempenho das aplicações, uma vez que essa etapa constitui a etapa mais cara de uma reconfiguração. Por essas razões, optou-se pela redistribuição simples, deixando as questões de balanceamento para serem decididas pelo escalonador de tarefas.

A operação de adição de uma instância segue o mesmo padrão de redistribuição de mensagens, com a diferença de que o local de mapeamento da instância sendo adicionada é definido, de maneira gulosa, como o ponto médio do maior intervalo entre

os mapeamentos de duas instâncias em execução. Essa estratégia visa particionar o espaço de processamento de maneira mais uniforme entre as instância de um filtro. A figura 4.7 ilustra a adição da instância I_3 a um filtro contendo 3 instâncias em execução.

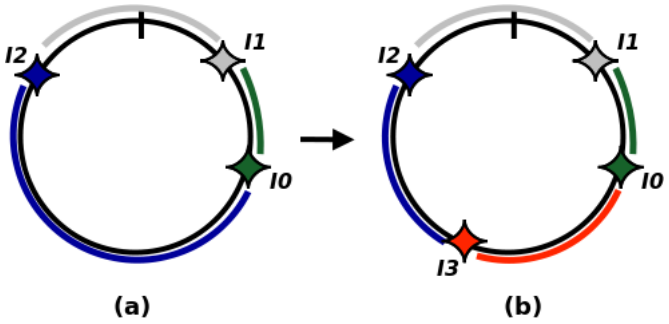


Figura 4.7: Redistribuição de mensagens após uma operação de adição de instância.

Como pode ser visto na figura, aproximadamente metade das mensagens antes direcionadas a I_2 passam a ser processadas pela nova instância, I_3.

Utilizando a abordagem de *hash* consistente para a redistribuição de mensagens, grande parte dessas não são afetadas e uma pequena porção, em média $\frac{1}{n+1}$ ou $\frac{1}{n-1}$, deve ser redirecionada a alguma instância que se torna responsável pelo seu processamento.

4.3.3 Particionamento Consistente de Estado

Como mencionado anteriormente, alguns filtros construídos em Watershed podem conter dados que devem ser armazenados durante o processamento. Esses dados, de importância para o domínio da aplicação, constituem o estado de um filtro. A funcionalidade de replicação dos filtros de processamento em instâncias idênticas tem como resultado o particionamento do estado do filtro, ou seja, os dados necessários ao processamento são distribuídos entre os processos que compõem o filtro.

Em particular, no caso de filtros que recebem dados por meio de fluxos rotulados, a consistência da computação é um problema inerente à função de mapeamento de mensagens e distribuição de dados processados. Isso ocorre porque os dados armazenados em cada uma das instâncias desses filtros carregam um significado semântico oriundo das mensagens que os geraram. Retomando o exemplo do contador de palavras, a contagem parcial das ocorrências de uma determinada palavra está armazenada em uma única instância e esse registro carrega a semântica das mensagens recebidas por tal instância contendo a palavra em questão. No caso de um remapeamento de mensagens, o novo destinatário das mensagens contendo essa palavra específica deve

ser o novo detentor de sua contagem parcial, de forma que o resultado da aplicação mantenha-se correto.

A estratégia utilizada para promover a consistência de filtros com estado é também baseada no conceito de *hash* consistente, utilizado para redistribuição de mensagens após uma reconfiguração. A alteração mais significativa está no fato de que a divisão do espaço de processamento entre as instâncias de um filtro é feita por meio de pequenos blocos de trabalho, possibilitando a manipulação de porções de dados independentes, como pode ser visto na figura 4.8.

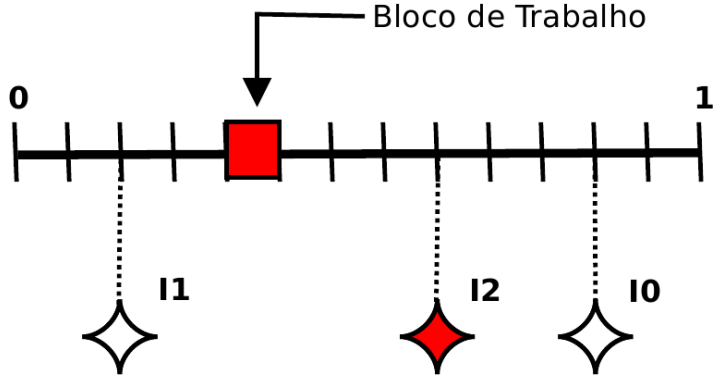


Figura 4.8: Subdivisão do espaço de processamento em blocos de trabalho para manutenção do estado consistente.

Por questões de apresentação, a circunferência de mapeamento é exibida em sua forma linearizada. A figura mostra a subdivisão do espaço de processamento entre as 3 instâncias de um filtro qualquer. A ideia é incorporar um segundo nível de divisão do estado do filtro. O primeiro nível é resultado direto da função de distribuição de mensagens entre as instâncias de um filtro. O segundo nível é construído com a divisão global do espaço de processamento em um número fixo de blocos de mesmo tamanho. Assim, quando uma partição do estado de um filtro é assinalado a uma determinada instância, essa instância passa a ser responsável por todos os blocos de trabalho dentro do intervalo assinalado. Como exemplo, o bloco de trabalho destacado na figura é atribuído à instância I_2.

No caso limite, o número de blocos de trabalho é igual ao número de rótulos diferentes que podem ser gerados pela função de rotulação. Neste trabalho definiu-se o número de blocos igual ao número máximo de instâncias que um filtro pode ter, que é um parâmetro de configuração do ambiente.

A estratégia utilizada tem como objetivo permitir a separação de dados processados, de forma que uma reconfiguração possa ser realizada sem perda de semântica para o filtro sendo reconfigurado. Nos casos de adição e remoção de instâncias, a re-

distribuição do estado é feita com base nos blocos de dados que devem ser movidos de uma instância para outra, sem que haja a necessidade de se separar os dados já processados. Obviamente, o posicionamento das instâncias de um filtro na circunferência de mapeamento é tal que fique exatamente entre dois blocos de trabalho, garantindo assim que nenhum bloco seja assinalado a mais de uma instância.

4.4 Implementação e Interface de Programação

O mecanismo de reconfiguração dinâmica compreende basicamente a implementação do protocolo de funcionamento, descrito na seção 4.2.2 e a implementação das estratégias de redistribuição de mensagens e consistência de estados descritas na seção anterior.

O protocolo de funcionamento foi implementado de forma direta, seguindo exatamente os passos descritos. A redistribuição de mensagens e particionamento de estados foram realizados da seguinte maneira. Primeiramente, a função de obtenção de rótulo `GetLabel` definida na seção 3.2.1 foi modificada para que não mais seja passado o número de instâncias do filtro como argumento. Isso porque com o mecanismo de reconfiguração, o programador de aplicação não deve elaborar sua função de rotulação com base no estado atual do filtro, mas sim com base unicamente no conteúdo da mensagem. A função modificada passa a ter a seguinte assinatura:

```
int GetLabel(Message&)
```

Três funções foram acrescentas à API de Watershed para a execução de filtros reconfiguráveis. As duas primeiras devem ser obrigatoriamente implementadas pelo programador de aplicação, pois são invocadas pelo ambiente sempre que necessário. A terceira função é exportada pela API do ambiente para que o programador possa ter acesso aos dados armazenados em determinada instância de filtro.

- `void SetState(State&)`: função de assinalamento de estado a uma instância de processamento. O ambiente de programação invoca essa função sempre que é necessário alterar o bloco de trabalho onde a instância deve escrever seus resultados ou ler dados consolidados.

- `State GetState()`: função de obtenção de estado de uma instância em execução. O ambiente invoca essa função sempre que o armazenamento do estado for necessário ou quando a troca de blocos de trabalho deva ser realizada.

- `StateIterator* GetStateIterator()`: função que retorna um iterador para o primeiro bloco de trabalho de responsabilidade de uma instância. A classe `StateIterator` possui métodos de obtenção dos blocos de trabalho em ordem contígua, além de métodos de controle para verificação do fim da lista e obtenção dos dados.

Internamente, o processo que implementa uma instância de filtro armazena uma lista de blocos de trabalho, que são exatamente os blocos assinalados por meio do *hash* consistente. Quando uma mensagem deve ser entregue a um filtro, essa mensagem é mapeada em algum bloco de trabalho, considerando o espaço global de processamento. O ambiente então verifica qual é a instância responsável pelo bloco em questão, direcionando a mensagem a ele. Essa verificação é realizada a um custo $O(\log n)$, onde n é o número de instâncias do filtro destino. Isso é possível porque o mapeamento das instâncias na estratégia de *hash* consistente é feito em um arranjo de acesso direto, ordenado pelo número do primeiro bloco de responsabilidade das instâncias. Quando a instância correta recebe a mensagem, utilizará as funções `GetState` e `SetState` para trocar o bloco de trabalho da instância, em resposta à mensagem que deve ser processada. Nos casos em que mensagens consecutivas são direcionadas a um mesmo bloco de processamento, essa troca não é realizada.

Todas as estratégias descritas neste capítulo aplicam-se a qualquer tipo de filtro que possa ser implementado no ambiente Watershed, tenha ou não fluxos de entrada sob a política de rotulação de mensagens.

Os algoritmos da figura 4.9 mostram a distribuição e o processamento de uma mensagem utilizando as modificações implementadas.

Como pode ser visto nos algoritmos apresentados, o processamento se dá como se houvessem mais instâncias do que de fato existem. Isso acontece porque uma única instância é responsável por vários blocos de trabalho e, para que todos esses blocos sejam consistentes, a chegada de uma mensagem deve ser tratada como previsto na visão lógica do programador.

Uma reconfiguração acarreta, inevitavelmente, a movimentação de dados entre as instâncias do filtro sendo reconfigurado (operações de adição e remoção) ou a movimentação de dados de uma instância sendo migrada de uma máquina para outra. Para realizar essa movimentações, o mecanismo realiza o *checkpoint* coordenado do filtro sendo reconfigurado imediatamente antes de executar a operação. Isso é feito por meio do armazenamento dos blocos de trabalho em memória secundária, utilizando um sistema de arquivos distribuído, como o NFS (*Network File System*). No

momento da redistribuição dos blocos, as instâncias destino fazem a carga desses dados, armazenando-os na estrutura interna de blocos de dados. Ambas as etapas são realizadas seguindo as orientações do *Gerente de Configuração*, que determina quais instância devem realizar *checkpoint* e quais devem carregar os dados armazenados para a continuação do processamento, caracterizando a etapa de *restart*.

```
Entrada: M: Mensagem a ser enviada. F: Filtro destino.
início
    se M.policy = broadcast então
        para cada f ∈ F faça
            Send(M, f)
        fim
    fim
    senão
        se M.policy = round robin então
            f ← NextToReceive()
        fim
        senão
            label ← GetLabel(M)
            block ← MapBlock(label, F)
            M.block ← block
            f ← MapInstance(block, F)
        fim
        Send(M, f)
    fim
fim
```

(a) Envio da mensagem.

```
Entrada: M: Mensagem recebida.
Dados: B: Conjunto de blocos sob responsabilidade dessa instância.
início
    block ← M.block
    lastBlock = LastProcessedBlock()
    se M.policy = broadcast então
        para cada b ∈ B faça
            B[lastBlock] ← GetState()
            SetState(b)
            lastBlock ← b
            Process(M)
        fim
    fim
    senão
        se M.policy = round robin então
            B[lastBlock] ← GetState()
            b ← (lastBlock + 1) % |B|
            lastBlock ← b
            SetState(b)
        fim
        senão
            se M.policy = labeled stream então
                se block ≠ lastBlock então
                    B[lastBlock] ← GetState()
                    lastBlock ← block
                    SetState(B[block])
                fim
            fim
        fim
        Process(M)
    fim
fim
```

(b) Recebimento da mensagem.

Figura 4.9: Algoritmos de envio e recebimento de mensagens utilizando subparticionamento de estado.

Capítulo 5

Avaliação Experimental

Esse capítulo descreve a avaliação experimental do mecanismo de reconfiguração dinâmica proposto no presente trabalho. Os experimentos são divididos em três etapas, caracterizadas pelas aplicações utilizadas em cada uma delas. Na primeira etapa foi desenvolvido um conjunto de filtros de processamento para a classificação de transações textuais utilizando algoritmos de aprendizagem supervisionada. Com essa aplicação, foram realizados testes de desempenho e eficácia do mecanismo.

Na segunda etapa foram realizados testes do custo das operações de reconfiguração em uma aplicação com estado. A aplicação utilizada nessa etapa é uma aplicação bem simples, cujo processamento não tem significado semântico, uma vez que o interesse está apenas em sua reconfiguração.

Por fim, a terceira etapa dos experimentos é realizada com o algoritmo de enumeração de conjuntos de itens frequentes denominado apriori. Nessa etapa foi avaliado o impacto de migrações de instâncias em uma aplicação com ciclo entre seus filtros e com a noção de terminação.

Essas aplicações foram escolhidas por representarem os tipos mais comuns de aplicações desenvolvidas para o ambiente Watershed.

Todos os experimentos foram conduzidos em um *cluster* com 12 máquinas interconectadas por um *Switch Gigabit Ethernet*. Cada máquina do *cluster* é equipada com um processador Intel ® Core ™ 2 CPU 6420 @2.13GHz, possui 2GB de memória RAM e executa o sistema operacional Linux. Em todos os experimentos realizados, cada máquina foi ocupada com no máximo uma instância de filtro, de forma que os recursos locais a essas máquinas não foram compartilhados. Além disso, cada experimento foi realizado 5 vezes, sendo os resultados apresentados graficamente como o valor médio das medições dessas execuções, acompanhados dos respectivos desvios padrões que são apresentados de maneira tabular. Em seguida é apresentada a descrição de cada uma

das aplicações escolhidas e os experimentos realizados com essas aplicações.

5.1 Classificação de Transações Textuais

A primeira aplicação é composta por um conjunto de 4 filtros. Essa aplicação realiza a classificação de transações textuais em classes de transações pré-definidas, de acordo com algoritmos de aprendizagem supervisionada. A figura 5.1 mostra a arquitetura da aplicação.

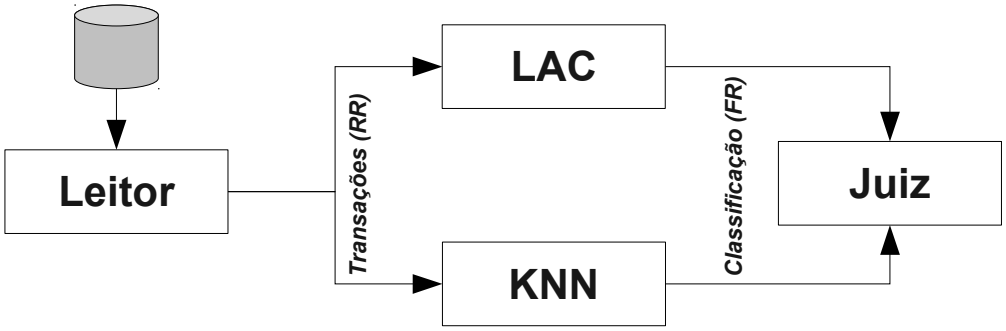


Figura 5.1: Arquitetura da aplicação de classificação de transações textuais.

O primeiro filtro da aplicação, denominado `Leitor`, realiza a leitura dos dados de um conjunto de teste armazenado em memória secundária. Esses dados estão armazenados no formato de um arquivo texto, onde cada linha representa uma transação que é composta pelo texto de um *tweet*. O conjunto de teste foi coletado em uma etapa *offline*. O filtro `Leitor` não possui estado e realiza a leitura quão rápido o sistema de arquivos local permita.

No segundo estágio de processamento existem dois filtros de classificação. Um desses filtros, denominado `KNN`, implementa o algoritmo k-NN, que, conforme descrito na seção 3.3.2.2, é uma técnica de associação que classifica uma transação t de acordo com as k transações mais próximas, conforme alguma métrica de similaridade, de uma base de treino T. O filtro `KNN` define que a distribuição de mensagens entre suas instâncias deve ser feita sob a política *Round-robin*, pois a classificação das transações dá-se de maneira independente de seus conteúdos. Os dados são recebidos do filtro `Leitor` no formato de cadeias de caracteres representando as transações. Esse filtro também não possui estado.

No mesmo estágio de classificação existe o filtro `LAC`, que implementa o algoritmo *Lazy Associative Classification* [Veloso et al., 2006], que é um algoritmo de classificação baseado em regras de associação geradas sob demanda para cada transação a ser classificada. Nesse algoritmo, somente regras úteis à classificação são geradas e essas

são armazenadas em uma estrutura de *cache* para posterior utilização, o que torna o algoritmo bastante eficiente. O filtro `LAC` também recebe mensagens do filtro `Leitor` segundo a política *Round-robin* no formato de cadeias de caracteres. O filtro `LAC` possui como estado a estrutura de *cache* utilizada para o armazenamento de regras de associação. Contudo, a utilização da política *Round-robin* torna a utilização do mecanismo de partição de estado desnecessário. Por isso, não foi utilizado nesse filtro.

Todas as instâncias de ambos os filtros do estágio de classificação realizam, no início da execução, a carga de um conjunto de treino com 100 transações, classificadas em uma etapa *offline*. Como saída, ambos os filtros enviam a classificação de uma transação utilizando a tupla `(id, classe, valor)`, que contém o identificador da transação, a classe atribuída pelo classificador e a nota dada pelo classificador como fator de decisão pela classe, respectivamente.

O último filtro dessa aplicação foi denominado `Juiz`, pois tem como função dar a classificação final para cada transação de acordo com as classificações realizadas pelo estágio anterior. Esse filtro possui uma estrutura de dados que armazena a primeira classificação de cada transação que chega a ele. Quando a segunda classificação de uma transação é recebida, o filtro toma a decisão sobre qual classe deve ser atribuída a ela, tendo como base as notas e classes das duas classificações. Após realizar a classificação final de uma transação, a entrada correspondente é removida da estrutura de dados. A estrutura que armazena a primeira classificação de cada transação constitui o estado do filtro em questão, que recebe os dados do estágio anterior utilizando a política de fluxo rotulado.

Após algumas medições preliminares, foi constatado que o filtro `LAC` possui a maior complexidade computacional, sendo um gargalo da aplicação. Por tal motivo, os experimentos realizados com essa aplicação envolvem reconfigurações aplicadas a esse filtro. O primeiro experimento utiliza um conjunto de teste com 1 milhão de transações a serem classificadas. Inicialmente foi carregada uma instância de cada filtro no ambiente Watershed. A cada 100 segundos de execução, uma operação de adição de uma instância foi aplicada ao filtro `LAC`, variando-se o número de instâncias de 1 a 9. A taxa de processamento foi monitorada na saída do filtro `Juiz` com periodicidade de 5 segundos. O gráfico da figura 5.2 mostra a variação da taxa de processamento ao longo do tempo.

Os resultados mostram que a taxa de processamento varia linearmente à medida que novas instâncias são adicionadas ao filtro `LAC` até a sexta reconfiguração (marca de 600 segundos). A partir de então, o filtro `KNN` torna-se o gargalo da aplicação e a adição de novas instância ao filtro `LAC` não surtem efeito na taxa de processamento.

Observando o gráfico, é possível identificar patamares entre reconfigurações. Es-

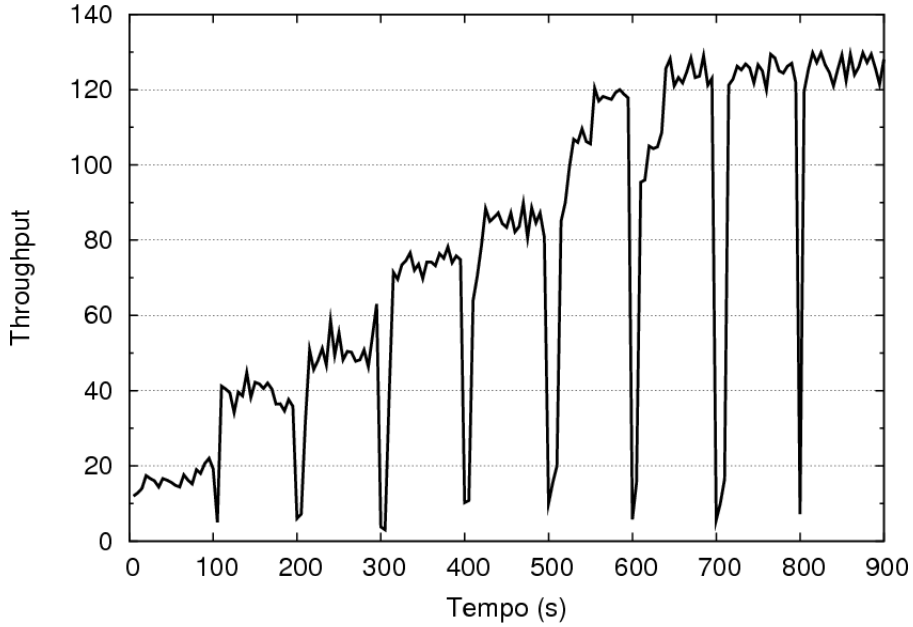


Figura 5.2: Taxa de processamento para adições sucessivas de instâncias na aplicação de classificação de transações textuais.

ses patamares indicam a taxa de processamento quase constante quando da ausência reconfigurações, o que é uma característica do ambiente Watershed. Todavia, as pequenas oscilações observadas têm origem na estratégia de controle de fluxo baseada em créditos, que favorece o processamento em rajadas.

O gráfico mostra também a redução drástica na taxa de processamento durante a transição entre configurações. Esse resultado era esperado, pois depois de uma sincronização entre o filtro sendo reconfigurado, seus produtores e seus consumidores, ocorre a adição da instância propriamente dita, suspendendo a execução do filtro por um tempo. O motivo pelo qual a taxa não chega a zero está no fato de que a amostragem é realizada com periodicidade de 5 segundos e, como a reconfiguração é rápida para filtros sem estado armazenado, algumas transações são processadas logo após a reconfiguração e antes da coleta dos dados de processamento.

No segundo experimento com a aplicação de classificação de transações, o objetivo é avaliar o ganho de desempenho da aplicação após reconfigurações aplicadas em diferente pontos de sua execução. Foram feitas reconfigurações no filtro `LAC`, dobrando-se o número de instâncias em cada fase do experimento. Os resultados para o processamento de 1 milhão de transações são mostrados no gráfico da figura 5.3.

Em uma primeira fase, executou-se a aplicação com 1 instância de cada filtro. Em seguida, uma nova execução foi iniciada, sendo que quando 75% das transações tinham sido processadas, uma reconfiguração foi aplicada, dobrando o número de instâncias

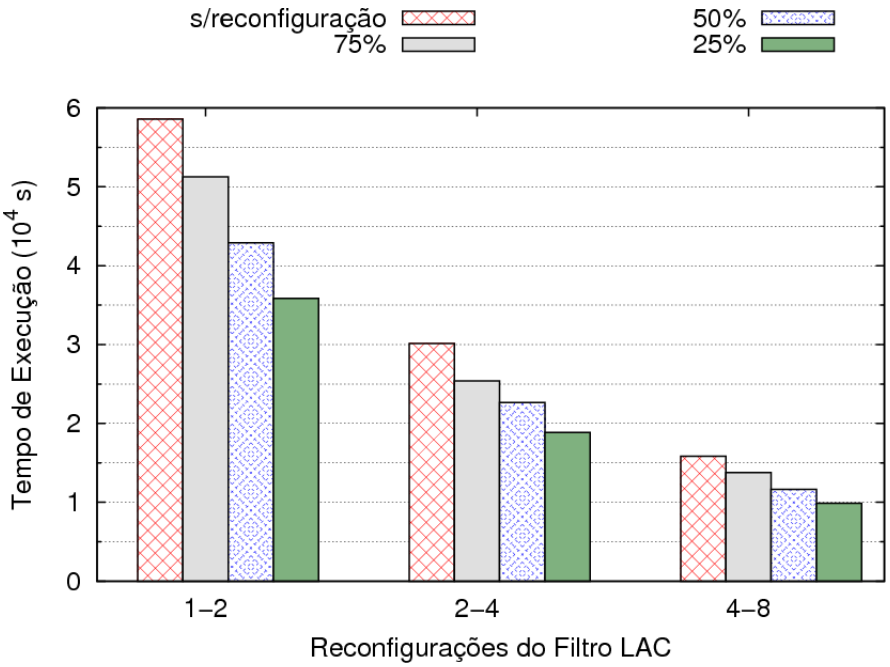


Figura 5.3: Tempo de execução para reconfigurações realizadas na aplicação de classificação de transações textuais

Tabela 5.1: Desvio padrão, em segundos, para o tempo de execução da aplicação de classificação de transações textuais para diferentes reconfigurações.

	Momento da Reconfiguração			
Ação	s/Reconf.	75%	50%	25%
1-2	110,97	16,09	70,27	47,78
2-4	267,92	44,24	123,77	67,71
4-8	138,93	60,35	53,38	15,02

do filtro `LAC`. O mesmo procedimento foi realizado nos pontos de progresso 50% e 25%. Os mesmos procedimentos dessa primeira fase são aplicados a duas outras fases, carregando o filtro `LAC` inicialmente com 2 e 4 instâncias, respectivamente.

Como pode ser visto no gráfico, o tempo de execução varia linearmente, considerando-se as reconfigurações realizadas no mesmo ponto de processamento.

Para as reconfigurações realizadas no ponto de progresso de 75%, o tempo de processamento foi reduzido para aproximadamente 87,5% do tempo original. Esse resultado vai de encontro com o valor teórico esperado, que pode ser obtido da seguinte maneira:

Para o processamento de 75% das transações, é necessário 75% do tempo gasto sem reconfigurações. Após a reconfiguração, é necessário processar os 25% restantes das

transações, o que é feito, idealmente, em 12,5% do tempo gasto sem a reconfiguração. Isso porque a reconfiguração em questão dobra o número de instâncias do filtro gargalo, o que reduz o tempo de processamento necessário pela metade. No final, tem-se o processamento de toda a base de transações em $75\% + 12,5\% = 87,5\%$ do tempo gasto originalmente. Similarmente, as reconfigurações feitas em 50% e 25% reduzem o tempo de processamento original da base para aproximadamente 75,0% e 62,5%.

A tabela 5.1 mostra o desvio padrão observado em cada um dos pontos do gráfico. Como pode ser facilmente identificado, a variação é muito pequena em relação aos tempos médios encontrados, o que demonstra a estabilidade do mecanismo diante das repetições do experimento.

Os resultados apresentados estão bem próximos dos valores teóricos esperados e mostram que o mecanismo de reconfiguração não tem impacto negativo significativo no desempenho das aplicações quando aplicado a filtros sem estado.

5.2 Processador de Números

Para avaliar o impacto do mecanismo de redistribuição de blocos de trabalho sobre o desempenho de aplicações construídas para executar em Watershed foi criada uma aplicação fictícia, denominada Processador de Números. Essa aplicação é composta por três filtros. O primeiro filtro, `F1`, gera números inteiros de maneira aleatória e os escreve em seu fluxo de saída. O segundo filtro, denominado `F2`, recebe os números produzidos por `F1` utilizando a política de fluxo rotulado e simplesmente escreve os dados de entrada em seu fluxo de saída. O filtro `F3` é o último filtro da aplicação, sendo responsável por contabilizar a quantidade de números pares e números ímpares que chegam até ele sob a política de distribuição alternada. A arquitetura dessa aplicação simples é ilustrada pela figura 5.4.

Figura 5.4: Arquitetura da aplicação de processamento de números.

O primeiro experimento tem como objetivo avaliar o custo das operações de reconfiguração em função da quantidade de dados armazenados em cada um dos blocos de trabalho da estratégia de consistência baseada em sub-particionamento do estado de um filtro.

A aplicação foi carregada no ambiente Watershed contendo 4 instâncias do filtro `F2` e 1 instância de cada um dos demais filtros. Variou-se o tamanho dos blocos de trabalho mantidos por `F2` entre 256KB e 1GB, dobrando-se os valores a cada nova execução. Os tempos gastos para cada uma das operações de reconfiguração em função do tamanho do bloco de dados são mostrados no gráfico da figura 5.5, sendo que o eixo x encontra-se na escala logarítmica de base 2. Os respectivos valores para o desvio padrão em cada ponto do gráfico são mostrados na tabela 5.2.

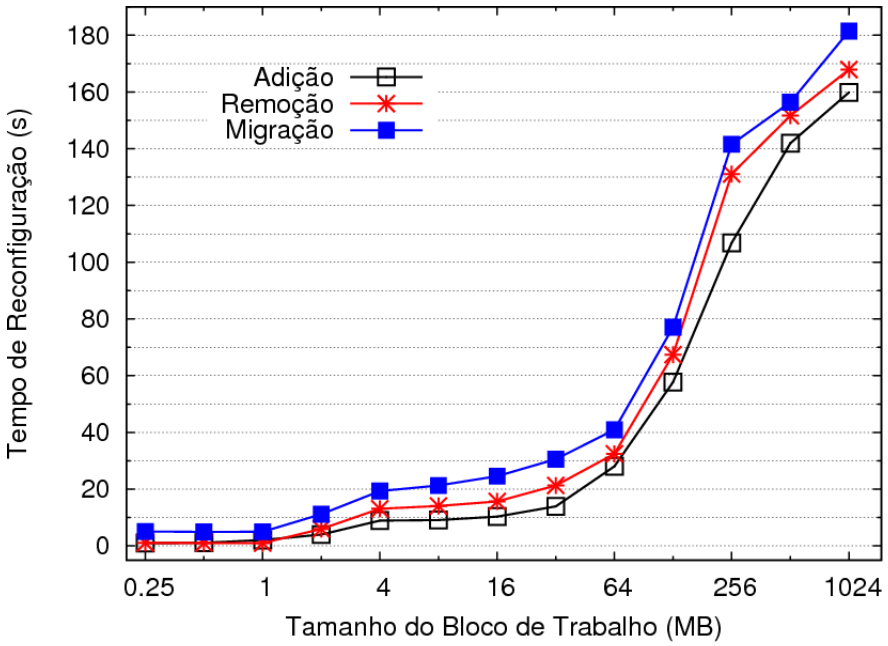


Figura 5.5: Tempo de reconfiguração em função do tamanho do bloco de trabalho.

Os dados obtidos nesse experimento mostram que as operações de reconfiguração apresentam custo significativo à medida que a quantidade de dados armazenados em um bloco de trabalho aumenta, chegando a variar linearmente com essa grandeza. O custo associado à reconfiguração reside basicamente nas operações de disco que devem ser feitas na fase de *checkpoint* e na fase de *restart* do filtro sendo reconfigurado. Uma observação importante é a ordem dos custos das operações. Como pode ser visto, a operação de migração mostrou-se a mais cara, seguida da operação de remoção e, por último, a operação de adição. A operação de adição é a que envolve a menor movimentação de dados para o caso específico desse experimento, uma vez que apenas cerca de 12,5% dos dados do estado compartilhado são movidos para o exemplo com 4 instâncias iniciais. Já a remoção de uma instância envolve a movimentação de cerca de 25% dos dados do estado para o exemplo, sendo uma operação mais dispendiosa. Teoricamente, o tempo gasto para uma remoção deveria ser o dobro do tempo gasto para uma adição, o que não foi confirmado pelo experimento. Uma análise mais deta-

Tabela 5.2: Desvio padrão, em segundos, para o tempo de reconfiguração em função do tamanho do bloco de trabalho.

Tamanho do	**Operação**		
Bloco (MB)	Adição	Remoção	Migração
0,25	0,16	0,23	0,36
0,50	0,16	0,19	0,51
1,00	0,27	0,16	0,68
2,00	0,19	0,34	0,51
4,00	0,23	0,33	0,77
8,00	0,25	0,40	1,58
16,00	0,47	0,47	0,82
32,00	0,29	0,67	0,99
64,00	0,48	0,62	1,93
128,00	1,21	1,15	0,95
256,00	1,08	1,56	3,17
512,00	2,71	1,76	2,71
1.024,00	3,42	2,73	2,05

lhada desse ponto constará na lista de trabalhos futuros. Já a operação de migração de processos constitui a operação mais cara, sendo a quantidade de dados movimentados a mesma da operação de remoção, com o acréscimo da criação do processo em uma máquina diferente.

A estabilidade do mecanismo também foi comprovada nesse experimento, uma vez que os dados coletados nas repetições não mostraram divergências significativas em relação à média calculada. Como exemplo, para a operação de adição de instância em um cenário com 1GB de dados em cada bloco de trabalho, foi observado um desvio padrão de 3,42 segundos, em relação a uma média de aproximadamente 160 segundos para a operação, o que corresponde, percentualmente, a cerca de 2,1%, que é um valor aceitável para os requisitos do ambiente.

Um segundo experimento foi realizado para verificar o tempo de reconfiguração de acordo com o número de instâncias do filtro sendo reconfigurado. Para isso, fixou-se o tamanho do bloco de trabalho em 16MB e variou-se o número de instâncias do filtro `F2` entre 1 e 8. Por ter apresentado o maior custo entre as operações, a migração foi escolhida para ser aplicada a cada uma das etapas desse experimento, que tem como resultado o gráfico da figura 5.6, acompanhado da tabela 5.3.

O gráfico mostra que, apesar de não ser um comportamento linear, o tempo de uma migração é reduzido com aumento do número de instâncias do filtro `F2`. Isso acontece porque quanto maior o número de instâncias, menor a quantidade de dados arma-

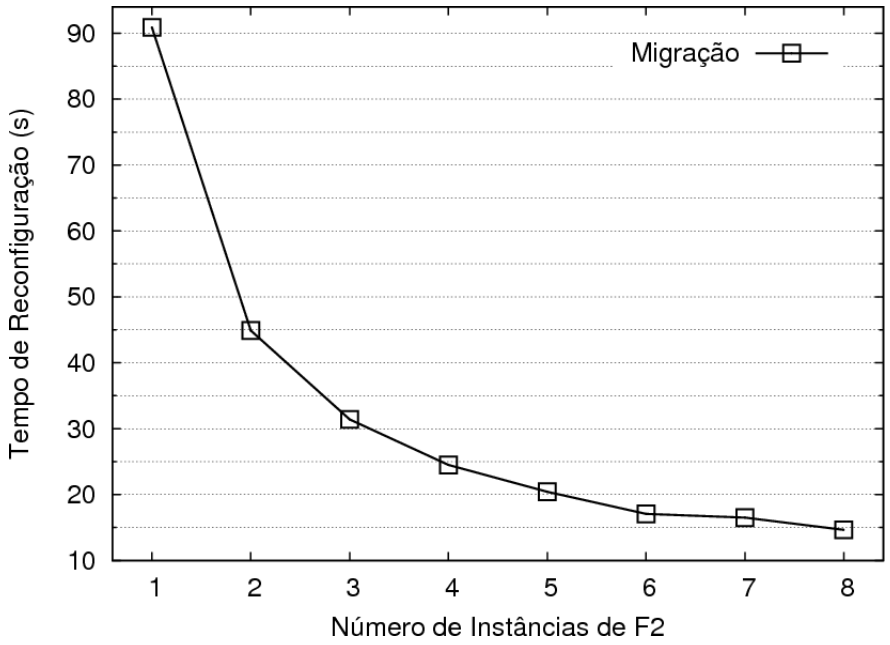


Figura 5.6: Tempo de reconfiguração em função do número de instâncias do filtro.

Tabela 5.3: Desvio padrão, em segundos, para o tempo de reconfiguração em função do número de instâncias do filtro sendo reconfigurado.

Número de Instâncias	**Desvio Padrão**
1	0,96
2	0,77
3	0,70
4	0,66
5	1,69
6	0,70
7	0,60
8	0,50

zenados em cada uma delas e, consequentemente, menor será o custo de movimentação de uma instância. Novamente, os dados coletados entre repetições do experimento não apresentam variações significativas em relação à média.

5.3 Algoritmo Apriori

Essa aplicação é exatamente a mesma descrita na seção 3.3.2.1. Para fins didáticos, algumas características da aplicação serão novamente descritas nesta seção. A figura 5.7 mostra a arquitetura da aplicação.

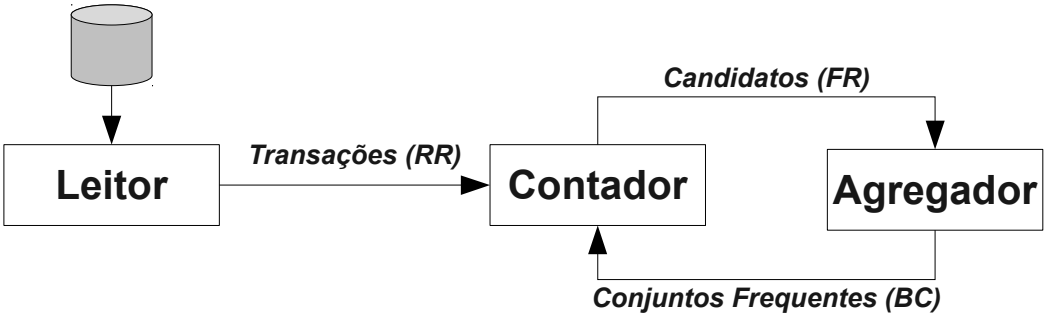


Figura 5.7: Arquitetura da aplicação Apriori.

A implementação é composta por três filtros: o `Leitor`, que realiza a leitura das transações da base de dados, o `Contador`, que gera candidatos, e o `Agregador`, que combina resultados parciais gerados pelo `Contador`. O filtro `Contador` gera candidatos por meio da contagem de ocorrências de cada atributo presente nas transações recebidas do `Leitor` via política *Round-robin*. Os pares contendo cada candidato e o número de ocorrências são enviados para `Agregador`, que faz a combinação das contagens parciais. Os atributos infrequentes são eliminados e o resultado final para os conjuntos de itens de tamanho 1 é obtido. Os conjuntos frequentes são informados ao filtro `Contador`, via *Broadcast*, para a geração dos candidatos de tamanho 2. O processo é repetido até que todos os conjuntos de itens possíveis sejam gerados.

Para esse experimento foram utilizadas 4 instâncias do filtro `Contador` e duas instâncias de cada um dos demais filtros. O filtro `Contador` foi utilizado como alvo das reconfigurações, uma vez que apresenta a maior complexidade computacional da aplicação. O estado desse filtro é constituído pela lista de conjuntos de itens a serem combinados para a geração de novos candidatos entre as iterações.

A aplicação foi executada sob 7 cenários distintos, sendo processadas 100 mil transações em cada um deles. O número de migrações de instâncias do filtro `Contador` foi variado entre 0 e 32. O tempo de execução médio e o respectivo desvio padrão para cada um dos cenários são mostrados no gráfico da figura 5.8 e na tabela 5.4, respectivamente.

O gráfico mostra que a utilização indiscriminada da operação pode ter um impacto importante no desempenho de uma aplicação. No caso do experimento, as migrações foram realizadas sem trazer um benefício para o desempenho geral da aplicação, ou seja, não houve migração de um servidor sobrecarregado para outro ocioso. Com isso, o custo da operação domina os resultados. É importante observar que a decisão de migração, tema não abordado nesse trabalho, deve ser tomada com base em dados de utilização de recursos, proximidade dos dados de entrada e outros fatores, de forma que

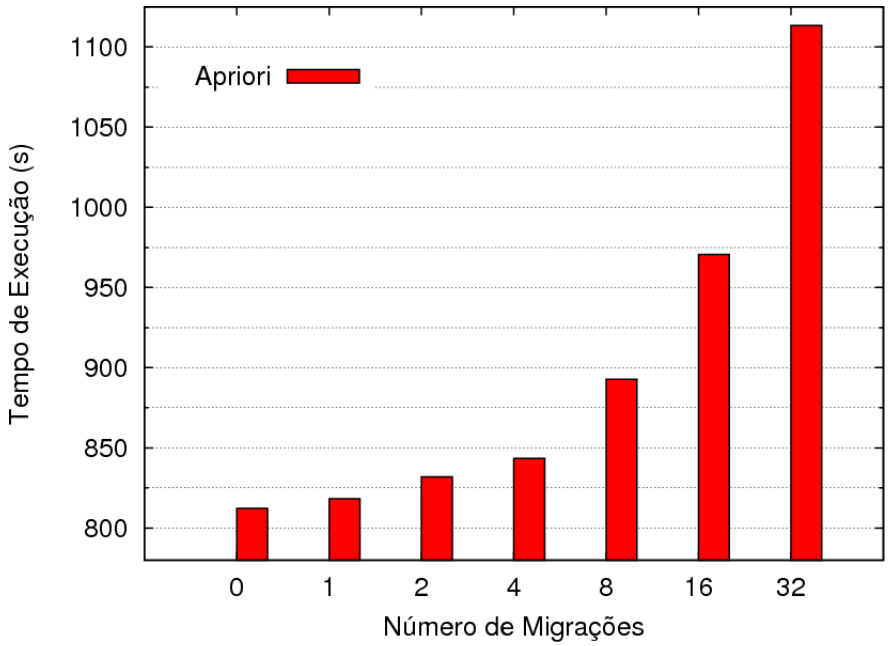


Figura 5.8: Tempo de execução em função do número de migrações de uma instância para a aplicação Apriori.

Tabela 5.4: Desvio padrão para o tempo de execução em função do número de operações de migração.

Número de Instâncias	**Desvio Padrão**
0	1,92
1	2,39
2	2,55
4	1,95
8	2,77
16	3,65
32	2,70

o custo da operação seja compensado pelo aumento na taxa de processamento do filtro sendo reconfigurado. Contudo, o acréscimo de tempo devido às operações de migração mostrou-se linear com relação ao número de operações realizadas, o que é um bom indicativo sobre o baixo *overhead* causado pelas etapas de sincronização necessárias à efetivação da operação.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste trabalho foi apresentado um mecanismo de auxílio à reconfiguração dinâmica para aplicações de processamento distribuído de fluxos de dados. Tal mecanismo foi implementado sobre o ambiente de execução Watershed, cuja arquitetura, modelo de programação e resultados experimentais são detalhados como parte do presente trabalho.

A estratégia utilizada consiste em uma nova forma de possibilitar operações de reconfiguração mantendo-se a consistência das aplicações reconfiguradas. Nessa estratégia, os processos em execução são considerados entidades sem semântica própria, sendo responsáveis apenas pelo processamento dos dados a eles assinalados. O foco da estratégia está nos dados a serem processados, não importando qual processo será designado para o trabalho.

Os resultados obtidos na avaliação experimental mostram que o mecanismo não apresenta custos significativos para aplicações sem estado. Todavia, o custo de movimentação de dados para aplicações com estado é considerável, sendo justificado pela utilização do sistema de arquivos distribuído como área de troca das partições de estado. Qualitativamente, o mecanismo apresenta alto grau de abstração, exigindo a implementação, por parte do programador de aplicação, de apenas dois métodos simples para sua utilização. Adicionalmente, o mecanismo tem aplicabilidade ampla, podendo ser empregado em qualquer aplicação desenvolvida para Watershed, sem restrições.

6.1 Objetivos Alcançados

A proposta apresentada neste trabalho de mestrado contempla grande parte dos requisitos elencados como fundamentais para um bom mecanismo de reconfiguração dinâmica. Foi comprovado, por meio de experimentos qualitativos, que as aplicações

reconfiguradas evoluem corretamente, apresentando os mesmos resultados que apresentam na ausência de reconfigurações. O mecanismo pode ser utilizado sem restrições em qualquer aplicação construída para Watershed, atingindo o requisito de aplicabilidade. Um grande esforço foi aplicado ao projeto do mecanismo para que esse seja pouco intrusivo e que exija pouca intervenção do programador de aplicação. Como resultado, chegou-se a uma arquitetura semi-transparente, que delega ao programador de aplicação a implementação de apenas duas funções relacionadas aos dados que devem ser mantidos entre configurações.

O requisito de assincronia foi parcialmente alcançado, uma vez que é necessário suspender a execução do filtro sendo reconfigurado, de seus produtores e de seus consumidores durante a operação. Contudo, os demais filtros das aplicações executando no ambiente continuam o processamento sem perdas de desempenho. O custo de aplicação de reconfigurações mostrou-se alto em relação ao esperado. Grande parte desse custo deve-se à abordagem simplória de utilização do sistema de arquivos distribuído como área de movimentação de dados entre instâncias de filtros. Consequentemente, a escalabilidade do mecanismo não foi comprovada de maneira irrefutável, sendo esse tipo de avaliação detalhada parte dos trabalhos futuros.

6.2 Trabalhos Futuros

O presente trabalho pode ser estendido explorando-se, dentre outros, os seguintes pontos:

- Avaliar de maneira mais detalhada o custo e a escalabilidade das operações de reconfiguração.

- Projetar e implementar a movimentação de dados em *background*, possibilitando que os filtros envolvidos em uma reconfiguração continuem o processamento durante a operação.

- Experimentar outros mecanismos de movimentação de dados, utilizando estruturas mais eficientes, como a compactação de dados.

- Investigar o custo da utilização do NFS como ferramenta de armazenamento de blocos de trabalho. É possível que o armazenamento local e a posterior transferência dos blocos de interesse possa melhorar o desempenho do mecanismo como um todo.

- Implementar novas aplicações que explorem de forma exaustiva o mecanismo proposto. Essas aplicações devem, de preferência, variar naturalmente a quantidade de dados armazenados no estado particionado.
- Integrar o mecanismo de auxílio a reconfiguração dinâmica ao escalonador de tarefas desenvolvido por outro membro do grupo de pesquisa.

6.3 Publicações

Ramos, T. L. A. S.; **Oliveira, R. S.**; Carvalho, A. P.; Ferreira, R. A. C.; Meira Jr., W.. *Watershed: A High Performance Distributed Stream Processing System.* Proceedings of the 23rd International Symposium on Computer Architecture and High Performance Computing, 2011, Vitória, Brazil.

Referências Bibliográficas

Agbaria, A. & Friedman, R. (2002). Virtual machine based heterogeneous checkpointing. In *IPDPS '02: Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium*, pp. 22–27. http://dx.doi.org/10.1109/IPDPS.2002.1015495.

Agbaria, A. M. & Friedman, R. (1999). Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *HPDC '99: Proceedings of the 8th International Symposium on High Performance Distributed Computing*, pp. 167–176. http://dx.doi.org/10.1109/HPDC.1999.805295.

Agrawal, R.; Imieliński, T. & Swami, A. (1993). Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pp. 207–216, New York, NY, USA. ACM. http://doi.acm.org/10.1145/170035.170072.

Amini, L.; Andrade, H.; Bhagwan, R.; Eskesen, F.; King, R.; Selo, P.; Park, Y. & Venkatramani, C. (2006). SPC: A distributed, scalable platform for data mining. In *DMSSP '06: Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms*, pp. 27–37, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1289612.1289615.

Beynon, M.; Chang, C.; Catalyurek, U.; Kurc, T.; Sussman, A.; Andrade, H.; Ferreira, R. & Saltz, J. (2002). Processing large-scale multi-dimensional data in parallel and distributed environments. *Parallel Computing*, 28(5):827–859. http://dx.doi.org/10.1016/S0167-8191(02)00097-2.

Beynon, M. D.; Kurc, T.; Catalyurek, U.; Chang, C.; Sussman, A. & Saltz, J. (2001). Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457--1478. http://dx.doi.org/10.1016/S0167-8191(01)00099-0.

Bhandarkar, M. A.; Kalé, L. V.; Sturler, E. d. & Hoeflinger, J. (2001). Adaptive load balancing for MPI programs. In *ICCS '01: Proceedings of the International Conference on Computational Science-Part II*, pp. 108–117, London, UK, UK. Springer-Verlag. http://dl.acm.org/citation.cfm?id=645456.654524.

Burge, W. H. (1975). Stream processing functions. *IBM Journal of Research and Development*, 19(1):12–25. http://dx.doi.org/10.1147/rd.191.0012.

Chandrasekaran, S.; Cooper, O.; Deshpande, A.; Franklin, M. J.; Hellerstein, J. M.; Hong, W.; Krishnamurthy, S.; Madden, S.; Raman, V.; Reiss, F. & Shah, M. A. (2003). TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR '03: Proceedings of the 1st Biennial Conference on Innovative Data Systems Research*, pp. 24–35, Asilomar, CA, USA.

Dean, J. & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In *OSDI '04: Proceedings of the 6th Symposium on Operating System Design and Implementation*, pp. 137–150, San Francisco, California, USA.

Desell, T.; Maghraoui, K. E. & Varela, C. A. (2006). Malleable components for scalable high performance computing. In *Proceedings of the HPDC '15 Workshop on HPC Grid programming Environments and Components (HPC-GECO/CompFrame)*, pp. 37–44, Paris, France. IEEE Computer Society.

Desell, T.; Maghraoui, K. E. & Varela, C. A. (2007). Malleable applications for scalable high performance computing. *Cluster Computing*, 10(3):323–337. http://dx.doi.org/10.1007/s10586-007-0032-9.

El Maghraoui, K.; Desell, T. J.; Szymanski, B. K. & Varela, C. A. (2007). Dynamic malleability in iterative MPI applications. In *CCGRID '07: Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*, pp. 591–598, Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/CCGRID.2007.45.

El Maghraoui, K.; Desell, T. J.; Szymanski, B. K. & Varela, C. A. (2009). Malleable iterative MPI applications. *Concurrency and Computation: Practice and Experience*, 21(3):393–413. http://dx.doi.org/10.1002/cpe.v21:3.

El Maghraoui, K.; Szymanski, B. K. & Varela, C. (2006). An architecture for reconfigurable iterative MPI applications in dynamic environments. In *PPAM '05: Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, pp. 258–271, Berlin, Heidelberg. Springer-Verlag. http://dx.doi.org/10.1007/11752578_32.

Feitelson, D. G.; Rudolph, L.; Schwiegelshohn, U.; Sevcik, K. C. & Wong, P. (1997). Theory and practice in parallel job scheduling. In *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pp. 1–34, London, UK, UK. Springer-Verlag. http://dl.acm.org/citation.cfm?id=646378.689517.

Ferreira, R. A.; Meira Jr., W.; Guedes, D.; Drummond, L. M. A.; Coutinho, B.; Teodoro, G.; Tavares, T.; Araujo, R. & Ferreira, G. T. (2005). Anthill: A scalable run-time environment for data mining applications. In *SBAC-PAD '05: Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, pp. 159–167, Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/CAHPC.2005.12.

Fireman, D.; Teodoro, G.; Cardoso, A. & Ferreira, R. (2008). A reconfigurable run-time system for filter-stream applications. In *SBAC-PAD '08: Proceedings of the 20th International Symposium on Computer Architecture and High Performance Computing*, pp. 149–156, Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/SBAC-PAD.2008.28.

Gabriel, E.; Fagg, G. E.; Bosilca, G.; Angskun, T.; Dongarra, J. J.; Squyres, J. M.; Sahay, V.; Kambadur, P.; Barrett, B.; Lumsdaine, A.; Castain, R. H.; Daniel, D. J.; Graham, R. L. & Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pp. 97–104, Budapest, Hungary.

Gedik, B.; Andrade, H.; Wu, K.-L.; Yu, P. S. & Doo, M. (2008). SPADE: The System S declarative stream processing engine. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pp. 1123–1134, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1376616.1376729.

Ghemawat, S.; Gobioff, H. & Leung, S.-T. (2003). The google file system. *Operating Systems Review (ACM SIGOPS)*, 37(5):29–43. http://doi.acm.org/10.1145/1165389.945450.

Gordon, M. I.; Thies, W. & Amarasinghe, S. (2006). Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS '06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 151–162, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1168857.1168877.

Huang, C.; Lawlor, O. & Kalé, L. V. (2003). Adaptive MPI. In *LCPC '03: Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, pp. 306–322, College Station, Texas.

Isard, M.; Budiu, M.; Yu, Y.; Birrell, A. & Fetterly, D. (2007). Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pp. 59–72, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1272998.1273005.

Kale, L. V. & Krishnan, S. (1993). CHARM++: a portable concurrent object oriented system based on C++. *SIGPLAN Notices*, 28(10):91–108. http://doi.acm.org/10.1145/167962.165874.

Karger, D.; Lehman, E.; Leighton, T.; Panigrahy, R.; Levine, M. & Lewin, D. (1997). Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pp. 654–663, New York, NY, USA. ACM. http://doi.acm.org/10.1145/258533.258660.

Koster, R.; Black, A. P.; Huang, J.; Walpole, J. & Pu, C. (2001). Infopipes for composing distributed information flows. In *Proceedings of the 2001 ACM Multimidia Workshop on Multimidia Middleware*, Ottawa, Canada.

Landin, P. J. (1965a). Correspondence between ALGOL 60 and church's lambda-notation: Part i. *Communications of the ACM*, 8(2):89–101. http://doi.acm.org/10.1145/363744.363749.

Landin, P. J. (1965b). A correspondence between ALGOL 60 and church's lambda-notations: Part ii. *Communications of the ACM*, 8(3):158–167. http://doi.acm.org/10.1145/363791.363804.

Liu, J. & Panda, D. K. (2004). Implementing efficient and scalable flow control schemes in mpi over infiniband. In *IPDPS '04: Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, USA. http://doi.ieeecomputersociety.org/10.1109/IPDPS.2004.1303193.

Macqueen, J. B. (1967). Some methods for classification and analysis of multivariate observations. In *Procedings of the Fifth Berkeley Symposium on Math, Statistics, and Probability*, volume 1, pp. 281–297. University of California Press.

Madden, S.; Shah, M.; Hellerstein, J. M. & Raman, V. (2002). Continuously adaptive continuous queries over streams. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pp. 49–60, New York, NY, USA. ACM. http://doi.acm.org/10.1145/564691.564698.

McCann, C. & Zahorjan, J. (1994). Processor allocation policies for message-passing parallel computers. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and Modeling of Computer Systems*, pp. 19–32, New York, NY, USA. ACM. http://doi.acm.org/10.1145/183018.183022.

Pu, C.; Schwan, K. & Walpole, J. (2001). Infosphere project: System support for information flow applications. *SIGMOD Record*, 30(1):25–34. http://doi.acm.org/10.1145/373626.373680.

Ramos, T. L. A. S.; Oliveira, R. S.; de Carvalho, A. P.; Ferreira, R. A. C. & Meira Jr., W. (2011). Watershed: A high performance distributed stream processing system. In *SBAC-PAD '11: Proceedings of the 23rd International Symposium on Computer Architecture and High Performance Computing*, pp. 191–198, Vitória, ES, Brazil. http://doi.ieeecomputersociety.org/10.1109/SBAC-PAD.2011.31.

Schneider, S.; Andrade, H.; Gedik, B.; Biem, A. & Wu, K.-L. (2009). Elastic scaling of data parallel operators in stream processing. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–12, Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/IPDPS.2009.5161036.

Sievert, O. & Casanova, H. (2004). A simple MPI process swapping architecture for iterative applications. *International Journal of High Performance Computing Applications*, 18(3):341–352. http://dx.doi.org/10.1177/1094342004047430.

Spencer, M.; Ferreira, R.; Beynon, M.; Kurc, T.; Catalyurek, U.; Sussman, A. & Saltz, J. (2002). Executing multiple pipelined data analysis operations in the grid. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pp. 1–18, Los Alamitos, CA, USA. IEEE Computer Society Press. http://dl.acm.org/citation.cfm?id=762761.762803.

Stellner, G. (1996). CoCheck: Checkpointing and process migration for MPI. In *IPDPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pp. 526–531. http://dx.doi.org/10.1109/IPPS.1996.508106.

Stephens, R. (1997). A survey of stream processing. *Acta Informatica*, 34:491–541. http://dx.doi.org/10.1007/s002360050095.

Sunderam, V. S. (1990). PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339. http://dx.doi.org/10.1002/cpe.4330020404.

Tanenbaum, A. S. & Steen, M. V. (2006). *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2nd edição.

Taura, K.; Kaneda, K.; Endo, T. & Yonezawa, A. (2003a). Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources. *SIGPLAN Notices*, 38(10):216–229. http://doi.acm.org/10.1145/966049.781533.

Taura, K.; Kaneda, K.; Endo, T. & Yonezawa, A. (2003b). Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources. In *PPoPP '03: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 216–229, New York, NY, USA. ACM. http://doi.acm.org/10.1145/781498.781533.

Teodoro, G.; Fireman, D.; Guedes, D.; Jr., W. M. & Ferreira, R. (2008). Achieving multi-level parallelism in the filter-labeled stream programming model. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pp. 287–294, Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/ICPP.2008.72.

Teodoro, G.; Hartley, T. D. R.; Catalyurek, U. & Ferreira, R. (2010). Runtime optimizations for replicated dataflows on heterogeneous environments. In *HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 13–24, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1851476.1851479.

Thies, W.; Karczmarek, M. & Amarasinghe, S. (2002). StreamIt: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pp. 179–196, London, UK. Springer-Verlag.

Thies, W.; Karczmarek, M.; Gordon, M.; Maze, D.; Wong, J.; Hoffmann, H.; Brown, M. & Amarasinghe, S. (2001). StreamIt: A compiler for streaming applications. Technical report, MIT-LCS Technical Memo TM-622, Cambridge, MA, USA.

Thies, W.; Karczmarek, M.; Sermulins, J.; Rabbah, R. & Amarasinghe, S. (2005). Teleport messaging for distributed stream programs. In *PPoPP '05: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 224--235, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1065944.1065975.

Utrera, G.; Corbalan, J. & Labarta, J. (2004). Implementing malleability on mpi jobs. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 215–224, Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/PACT.2004.18.

Vadhiyar, S. & Dongarra, J. (2003). SRS - a framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(2):291–312.

Veloso, A.; Meira Jr., W. & Zaki, M. J. (2006). Lazy associative classification. In *ICDM '06: Proceedings of the 6th International Conference on Data Mining*, pp. 645–654, Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/ICDM.2006.96.

Witten, I. H. & Frank, E. (2002). Data mining: Practical machine learning tools and techniques with java implementations. *ACM SIGMOD Record*, 31(1):76–77.