UM MECANISMO DE PERSISTÊNCIA PARA UM AMBIENTE DE PROCESSAMENTO DE FLUXOS DE DADOS

ANA PAULA DE CARVALHO

UM MECANISMO DE PERSISTÊNCIA PARA UM AMBIENTE DE PROCESSAMENTO 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

Agosto de 2012

de Carvalho, Ana Paula
M1234x Um Mecanismo de Persistência para um Ambiente de Processamento de Fluxos de Dados / Ana Paula de Carvalho. — Belo Horizonte, 2012
xxii, 60 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Renato Antônio Celso Ferreira

1. Persistência de Fluxos de Dados. 2. Computação de Alto Desempenho. 3. Processamento de Fluxos de Dados. I. Título.

CDU 100.0*01.10

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

FOLHA DE APROVAÇÃO

Um mecanismo de persistência para um ambiente de processamento de fluxos de dados

ANA PAULA DE CARVALHO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

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

PROF. ALBERTO HENRIQUE FRADE LAENDER
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, 17 de agosto de 2012.

Dedico este trabalho aos meus pais, Paulo e Berenice, a minha irmã, Marília, ao meu irmão, Marcelo e ao meu namorado Bruno.

Agradecimentos

Meu primeiro "Obrigada!"é destinado a minha família: Paulo (sempre presente), Berenice, Marcelo e Marília. Obrigada pelo apoio, incentivo e carinho.

Agradeço também a Rodrigo Oliveira, Luiz Ramos, Andrei Alvares, Moisés Pereira, Thatyene e principalmente Bruno Leite. Vocês foram fundamentais para realização deste trabalho.

Finalmente agradeço a meu orientador Renato Ferreira pela atenção e ensinamentos.

"Uma jornada de mil milhas começa com um único passo."
(Lao-Tzu)

Resumo

A contínua evolução da tecnologia nas variadas áreas do conhecimento vem propiciando que volumes cada vez maiores de dados estejam disponíveis. Por isso hoje, mais do que nunca, existe uma demanda real por aplicações capazes de processar grandes volumes de dados. Em geral, essas aplicações precisam executar, com alto desempenho, algoritmos computacionalmente intensivos que processam fluxos de dados. Diversas dessas aplicações também requerem que fluxos de dados sejam persistidos, principalmente pelas seguintes razões: i) possibilitar o rastreamento das transformações realizadas nos dados, ii) permitir que os dados sejam analisados futuramente e iii) reprocessar os dados em casos de falha. O objetivo geral do presente trabalho é contribuir no projeto e implementação do Watershed, um ambiente de execução de alto desempenho que provê abstrações para o desenvolvimento de aplicações distribuídas que processam fluxos de dados massivos. Para tanto, propõe-se nesta dissertação um mecanismo de persistência de fluxos de dados capaz de se acoplar ao Watershed. O ambiente de execução implementa o modelo de programação filtro-fluxo, dessa forma cada aplicação é decomposta em módulos de processamento que se comunicam por canais denominados fluxos. Algumas características diferem o Watershed da maioria dos ambientes/sistemas descritos na literatura, como: suporte ao desenvolvimento e execução de aplicações com topologia dinâmica, suporte a execução simultânea de múltiplas aplicações e possibilidade de compartilhamento de resultados intermediários entre diferentes aplicações. O mecanismo de persistência de fluxos proposto torna o Watershed um ambiente mais genérico e flexível, uma vez que possibilita que os módulos de processamento executem em diferentes períodos de tempo, tendo disponível para consumo todos os dados anteriormente produzidos, além dos dados atuais. O mecanismo também é distribuído, provê transparência no armazenamento dos dados, suporte a manipulação de dados semiestruturados e fornece recursos para que um módulo de processamento filtre de um fluxo apenas as unidades de dados, atuais ou históricas, de seu interesse. Nos experimentos realizados o impacto do mecanismo de persistência no tempo de execução das aplicações foi de no máximo 13%.

Palavras-chave: Persistência de fluxos de dados, computação de alto desempenho, processamento de fluxos de dados.

Abstract

The continuous evolution of technology in several areas of knowledge shows that increasing volumes of data are available. So today there exist, more than ever, a real demand for applications able to process large bodies of data. In general, these applications need to run at a high performance, intensive computationally algorithms that process data streams. Several of these applications also require that data streams are persisted, mainly for the following reasons: i) to enable the tracing of the transformations performed in the data, ii) to allow that the data analyzed in the future and iii) to reprocess the data in case of failure. The overall objective of this work is to contribute in the design and implementation of Watershed, a high-performance execution environment that provides abstractions for the development of distributed applications that process massive data streams. With that aim, we propose in this dissertation a data streams persistence mechanism that can be coupled to the Watershed. The execution environment implements the filter-stream programming model, so each application is decomposed into processing modules that communicate through channels called streams. Some features differ the Watershed from most environments/systems described in the literature, such as: support to the development and implementation of applications with dynamic topology, support the simultaneous execution of multiple applications and the possibility of intermediate results shared between among applications. The persistence mechanism proposed makes the Watershed a more general and flexible environment, since it enables that processing modules run at different time periods, having all data previously produced available for consumption, in addition to the current data. The mechanism is also distributed, it provides transparency in data storage, it supports semi-structured data handling and it provides resources for which a processing module filter from a stream uses only the units data, current or historical(stored), of its interest. In the experiments the impact of the persistence mechanism in the execution time of the applications was up to 13%.

Keywords: Persistence of data streams, high-performance computing, data stream

processing.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

O advento da Web 2.0 possibilitou que a Internet se tornasse uma das principais fontes de dados da atualidade. A Web 2.0 forneceu aos usuários não só a possibilidade de receber, mas também de adicionar conteúdo a Internet por meio de blogs, *wikis*, entre outros. Aliado a isso, a contínua evolução da tecnologia nas variadas áreas do conhecimento vem propiciando que volumes cada vez maiores de dados estejam disponíveis.

Muitas fontes produzem dados continuamente, um exemplo é o Twitter[1], que desde 2011 gera cerca de 1 bilhão de tweets por semana e chega até a atingir a marca de 4 mil *tweets*[2] por segundo quando ocorrem eventos de impacto [Lin et al., 2011]. Sequências de dados com ordenação temporal podem ser definidas como fluxos de dados [Gedik et al., 2008]. Exemplos de fluxos de dados incluem estatísticas de tráfego de rede, dados coletados por sensores, dados gerados por redes sociais e outros serviços na Internet.

Atualmente é crescente a demanda por aplicações capazes de processar fluxos de dados. Diversas áreas estão utilizando esse tipo de aplicação, exemplos incluem aplicações para engenharia de tráfego e segurança de redes de computadores [Chakravarthy & Jiang, 2009], aplicações de monitoramento de dados de sensores [Carney et al., 2002], aplicações médicas [Blount et al., 2010] e muitas outras. Essas aplicações, além de lidar com grandes volumes de dados, podem também apresentar computação intensiva. Em virtude disso, a comunidade científica tem investigado maneiras para atingir alto desempenho no processamento de fluxos de dados [Amini et al., 2006; Gedik et al., 2008; Neumeyer et al., 2010].

[1] Twitter: `www.twitter.com`

[2] *Tweets* são mensagens de no máximo 140 caracteres

1.1 Motivação

Hoje existe, mais do que nunca, uma demanda real por aplicações capazes de processar grandes volumes de dados [Kargupta & Han, 2009; Migliavacca et al., 2010; Gomide et al., 2011]. Em geral essas aplicações precisam executar com alto desempenho algoritmos computacionalmente intensivos que processam fluxos de dados. Muitas aplicações também requerem que os fluxos de dados sejam persistidos pelos seguintes motivos:

- Reprocessar os dados - possibilitar que em caso de falha, os dados possam ser reprocessados;
- Possibilitar a análise futura de dados atuais - os usuários podem desenvolver aplicações para processar dados que foram gerados antes delas começarem a executar;
- Rastrear as transformações realizadas nos dados - algumas aplicações precisam manter o histórico dos dados que foram processados ou o rastreamento das transformações realizadas nos dados (*data provenance*) [Moreau, 2010] para, por exemplo, possibilitar a avaliação da qualidade e validade dos resultados dos processamentos;

Um exemplo de sistema com aplicações deste tipo é o Observatório da Web[3]. Ele é um projeto de pesquisa pioneiro do InWeb (Instituto Nacional de Ciência e Tecnologia para a Web). Seu principal objetivo é coletar informações de diversas fontes na Web (redes sociais, blogs, portais, etc) e sintetizar e mostrar tais informações aos usuários na forma de indicadores e metáforas visuais. Para atingir seu objetivo, o Observatório da Web processa um gigantesco volume de dados utilizando diversas técnicas de mineração de dados [Santos et al., 2010]. Para permitir a realização de análises futuras e para fins de histórico, todos os dados iniciais e intermediários do Observatório da Web são armazenados em bancos de dados. Vale ressaltar que fica a cargo dos desenvolvedores de aplicações armazenar e recuperar esses dados.

Conforme já descrito, aplicações que processam grandes volumes de dados precisam executar com alto desempenho. Em vários casos existe também a necessidade de persistência dos dados, de forma que essa persistência não impacte negativamente no desempenho. Em face do exposto, se faz necessário o desenvolvimento de sistemas que permitam o desenvolvimento de aplicações de processamento de fluxos de dados, de forma a prover alto desempenho e à persistência de dados, além de abstrair as dificuldades inerentes a esse tipo de desenvolvimento.

[3]Observatório da Web: `http://observatorio.inweb.org.br/`

1.2 Objetivos

O Watershed [Ramos et al., 2011] é um ambiente de execução distribuído e de alto desempenho para processamento de fluxos de dados, que está sendo projetado e desenvolvido por um grupo de pesquisa do laboratório e-SPEED (*e-commerce System Performance Evaluation and Experimental Development*) da UFMG (Universidade Federal de Minas Gerais). O objetivo geral do presente trabalho é contribuir no projeto e implementação do Watershed. Para tanto propõe-se nesta dissertação um mecanismo de persistência de fluxos de dados para o Watershed que armazena os dados gerados pelos módulos de processamento e possibilita que os módulos consumam tanto dados atuais quanto dados históricos (armazenados). Os desafios relacionados à persistência de fluxos de dados incluem: i) Lidar com armazenamento e recuperação de grandes volumes de dados semiestruturados de forma eficiente; ii) Integrar o mecanismo de persistência ao ambiente de execução, de modo que seja minimizado o impacto no desempenho das aplicações.

1.3 Contribuições

Considera-se como principal contribuição deste trabalho o projeto, documentação e desenvolvimento de um mecanismo de persistência de fluxos de dados para o Watershed. O Watershed é um ambiente de execução que possui algumas características que o diferenciam da maioria dos demais ambientes/sistemas, como apoio ao desenvolvimento e execução de aplicações com topologia dinâmica, apoio à execução simultânea de múltiplas aplicações e possibilidade de compartilhamento de resultados intermediários entre diferentes aplicações. O mecanismo de persistência de fluxos de dados torna o Watershed um ambiente mais genérico e flexível, uma vez que possibilita que os módulos de processamento executem em períodos de tempos diferentes, mantendo disponível para consumo todos os dados anteriormente produzidos, além, é claro, dos dados atuais. Para o usuário do Watershed, o mecanismo de persistência abstrai o armazenamento e recuperação dos fluxos de dados históricos, que é uma tarefa tediosa, consome tempo e está sujeita a erros.

A seguir são descritas algumas das principais características do mecanismo de persistência de fluxos de dados proposto:

- Como o Watershed é um ambiente de execução para aplicações de processamento de fluxos de dados, o mecanismo de persistência lida com grandes volumes de dados;

- O Watershed é um ambiente genérico, assim as aplicações que executam nele podem lidar com dados semiestruturados. Por essa razão, o mecanismo de persistência possibilita o armazenamento e manipulação de dados semiestruturados;
- O mecanismo de persistência permite processamento distribuído;
- Para tornar o Watershed um ambiente mais flexível, o mecanismo de persistência possibilita à criação e execução de consultas personalizadas que permitem que um módulo de processamento filtre de um fluxo apenas os dados, atuais ou históricas de seu interesse.

1.4 Organização do Texto

O restante deste trabalho foi dividido em seis capítulos, organizados da seguinte forma:

Capítulo 2 [Trabalhos Relacionados] São discutidos alguns dos principais trabalhos relacionados às áreas de processamento e persistência de fluxos de dados;

Capítulo 3 [Watershed] Apresenta uma descrição detalhada do ambiente de execução Watershed;

Capítulo 4 [Persistência de Fluxos de Dados] Detalha o funcionamento e a implementação do mecanismo de persistência de fluxos de dados do Watershed;

Capítulo 5 [Avaliação Experimental] Apresenta os experimentos e discute os resultados obtidos;

Capítulo 6 [Conclusão e trabalhos futuros] Apresenta as conclusões do trabalho e indica possíveis caminhos para continuidade do mesmo.

Capítulo 2

Trabalhos Relacionados

Neste capítulo são discutidos alguns dos principais trabalhos relacionados às áreas de processamento e persistência de fluxos de dados. Especificamente, a Seção 2.1 apresenta trabalhos da área de processamento de fluxos de dados, a Seção 2.2 discute alguns trabalhos relacionados a persistência de fluxos de dados e, por fim, a Seção 2.3 descreve as principais características do MongoDB [Chodorow & Dirolf, 2010; Banker, 2011], um sistema de gerência de bancos de dados orientados a documentos que é utilizado no mecanismo de persistência proposto nesta dissertação.

2.1 Processamento de Fluxos de Dados

Uma forma para se obter eficiência no processamento de volumes massivos de dados é utilizar um conjunto de computadores para resolver problemas relacionados a uma dada aplicação. Várias abordagens têm sido propostas para atingir essa eficiência e prover abstrações para tornar mais fácil o desenvolvimento de aplicações paralelas e distribuídas. Isso motivou o surgimento de vários sistemas para processamento de fluxos de dados [Cugola & Margara, 2012]. A seguir são apresentados quatro dos principais sistemas de processamento de fluxos de dados descritos na literatura, DataCutter, Anthill, S4 e System S.

2.1.1 DataCutter

DataCutter [Andrade et al., 2001; Beynon et al., 2001, 2002] é um arcabouço para exploração e análise de dados científicos em ambientes distribuídos e heterogêneos. Ele implementa o modelo de programação filtro-fluxo (*filter-stream*). Nesse modelo os componentes de uma aplicação intensiva em dados são representados como conjuntos

de filtros. As trocas de dados entre os filtros são feitas por meio de canais unidirecionais chamados de fluxos.

No DataCutter são alcançadas duas formas de paralelismo: de dados e de tarefas. O paralelismo de tarefas é obtido através da execução concorrente dos filtros das aplicações como um *pipeline*. O paralelismo de dados é alcançado devido à criação de cópias transparentes dos filtros, dessa forma os dados podem ser divididos e processadados paralelamente pelas cópias dos módulos.

2.1.2 Anthill

O Anthill [Ferreira et al., 2005] é um ambiente de execução que, assim como o DataCutter, também implementa o modelo de programação filtro-fluxo. As abstrações fornecidas pelo Anthill tornam possível explorar o paralelismo de tarefas e de dados. Além disso, ele introduz um terceiro tipo de paralelismo: assincronia entre as iterações de aplicações cíclicas. O Anthill implementa manutenção de estado distribuído e um algoritmo para detecção de terminação que facilita o desenvolvimento de aplicações que possuem ciclos, visto que nesses casos a terminação pode ser complexa.

No Anthill foi implementada uma política de roteamento de mensagens denominado fluxo rotulado (*labeled stream*). Essa política possibilita que as messagens sejam encaminhadas para cópias específicas do filtro receptor.

A topologia das aplicações Anthill são definidas estaticamente por meio de um arquivo de configuração escrito utilizando a linguagem XML. Nesse arquivo são especificadas as máquinas do ambiente de execução, os filtros da aplicação e suas ligações, que são definidas como fluxos unidirecionais.

A seguir são descritos alguns trabalhos que propuseram extensões para o Anthill. Fireman et al. [2008] propõem um protocolo de suporte à adição dinâmica de instâncias em tempo de execução no Anthill. Além disso, eles propõem um módulo chamado Estado Global Distribuído, cujo objetivo principal é abstrair do programador detalhes relacionados a redistribuição do estado das aplicações.

[Teodoro et al., 2008] propõem o Anthill orientado a eventos que permite que os recursos disponíveis dentro de um nodo de computação sejam melhor utilizados, o que é importante principalmente para exploração de ambientes *multi-core*. No Anthill orientado a eventos ao invés dos programadores escreverem um laço de processamento dos dados para ser invocado uma única vez, eles fornecem funções para processar uma única unidade de dado por vez. Essas funções são invocadas quando há dados disponíveis para processamento. O ambiente é responsável por realizar o escalonamento e controle de dependências.

[Teodoro et al., 2010] apresentam uma extensão do Anthill orientado a eventos para ambiente heterogêneos. Nessa abordagem são utilizados tratadores de eventos para dispositivos de processamento heterogêneos implementados pelos usuários. O Anthill, em tempo de execução, utiliza os processadores disponíveis para executar os eventos de forma concorrente.

2.1.3 S4

S4 (*Simple Scalable Streaming System*) [Neumeyer et al., 2010] é uma plataforma distribuída de processamento de fluxos de dados desenvolvida pela Yahoo!. S4 foi projetado para lidar com aplicações para mineração de dados e algoritmos de aprendizado de máquina. O projeto do S4 é derivado de uma combinação do MapReduce e do modelo Atores (*Actors model*) [Agha, 1986].

As computações no S4 são realizados por unidades básicas chamadas elementos de processamento. A comunicação entre os elementos de processamento é feita por meio de transmissões de mensagens na forma de eventos de dados. Os eventos de dados são sequências de elementos na forma de pares chave/atributo. As chaves dos eventos são utilizadas para rotear os dados para os diferentes elementos de processamento. Os elementos de processamento consomem os eventos e em seguida publicam os resultados ou emitem novos eventos que podem ser consumidos por outros elementos.

2.1.4 System S

System S [Amini et al., 2006; Gedik et al., 2008] é um *middleware* de processamento de fluxos de dados, distribuído e de larga escala, desenvolvido pela IBM Research. Ele foi projetado para dar apoio a aplicações de mineração de fluxos de dados.

No System S, cada aplicação é composta de múltiplos Elementos de Processamento que são distribuídos nos nodos de computação e conectados entre si por meio de fluxos de dados, formando um grafo. Para programar as aplicações é utilizada uma linguagem chamada Spade [Gedik et al., 2008].

O System S permite a composição de aplicações com topologia dinâmica e a execução simultânea de múltiplas aplicações que podem interagir.

2.1.5 Comparação com Watershed

As seguintes características diferem o Watershed dos três primeiros sistemas: i) o Watershed permite o desenvolvimento e execução de aplicações com topologia dinâmica, ii) o Watershed possibilita que diversas aplicações executem simultaneamente e com-

partilhem módulos de processamento. O Watershed e o System S possuem várias características em comum, inclusive as citadas anteriormente. Uma das diferenças entre eles é que System S é um produto da IBM, comercialmente chamado InfoSphere Streams[1], e o Watershed é um ambiente de código aberto.

2.2 Persistência de Fluxos de Dados

Nesta seção são descritos alguns trabalhos relacionados que lidam com persistência de fluxos de dados.

Hildrum et al. [2008], membros do grupo de pesquisa do projeto System S, abordam problemas relacionados ao armazenamento de fluxos de dados em sistemas de processamento de fluxos distribuídos e de larga escala. Esse trabalho foca na otimização da colocação de dados dentro de um subsistema de armazenamento distribuído. Os principais objetivos do subsistema são: i) manter os dados com maior valor global; ii) balancear a carga de leitura para o sistema de arquivos. Os autores apresentam uma abordagem para remover automaticamente os dados armazenados na medida que novos dados vão chegando. A abordagem apresentada trata os dados de forma diferente considerando a sua importância para o sistema global. Isso é feito definindo-se para cada dado uma função que descreve seu valor projetado sobre o tempo. Assim que novos dados vão chegando o subsistema de armazenamento remove os dados armazenados que posssuem o menor valor corrente.

Nesse artigo é informado que o subsistema de armazenamento ainda não foi integrado ao System S e que para realizar essa integração uma série de adaptações e desenvolvimentos devem ser realizados. O presente trabalho lida justamente com alguns dos desafios da integração de um mecanismo de persistência em um ambiente de processamento de fluxos de dados. Um importante trabalho futuro a ser realizado é lidar com o problema de remover automáticamente os dados armazenados na medida que novos dados vão chegando.

O SMS (*Storage Manager for Streams*) [Botan et al., 2009] é um gerenciador de armazenamento de propósito geral para sistemas de gerenciamento de fluxos de dados, que desacopla a gerencia do armazenamento dos fluxos de dados do mecanismo de processamento. Segundo os autores, um sistema de processamento de fluxos de dados completo pode ser formado por dois componentes: o mecanismo de processamento de fluxos de dados e o SMS. O mecanismo de processamento de fluxos realiza o processamento das consultas e utiliza o SMS para executar todas as tarefas relacionadas ao ar-

[1]IBM InfoSphere Streams: `http://www-01.ibm.com/software/data/infosphere/streams/`

mazenamento. Por meio de uma interface, o mecanismo de processamento de fluxos de dados comunica para o SMS um conjunto de parâmetros que descreve as propriedades das instâncias de armazenamento que devem ser criadas. A lista de parâmetros que podem ser utilizados foi obtida através de análises que identificaram os principais requisitos de armazenamento. O SMS permite a criação de instâncias de armazenamento personalizadas que representam a implementação das propriedades requeridas.

O SMS lida apenas com armazenamento de fluxos de dados em ambientes não distribuídos, diferente do mecanismo de persistência proposto neste trabalho, que é acoplado ao Watershed e gerencia a manipulação e armazenamento de fluxos de dados, atuais e históricos, em um ambiente paralelo e distribuído.

PTS (*Persistent Temporal Streams*) [Hilley & Ramachandran, 2009] é um sistema que provê abstrações que abrangem transporte, manipulação e armazenamento de fluxos de dados. No PTS os fluxos temporais são representados por canais, que são estruturas de dados distribuídas. Cada canal possui uma sequência de itens de dados indexados pela informação de tempo. As aplicações podem interagir com os canais por meio de operações *get* e *put*. A operação *put* coloca um item de dados com *timestamp* no fluxo. A operação *get* recupera todos os itens, atuais ou persistidos, que estão em um determinado intervalo de tempo.

A recuperação de dados no PTS é baseada apenas em consultas relacionados a tempo. O mecanismo de persistência do Watershed permite recuperar unidades de dados gerados a partir de uma determinada data, além de possibilitar que as unidades de dados dos fluxos possam ser filtradas por meio de consultas sofisticadas especificadas pelos usuários.

2.3 MongoDB

MongoDB[2] [Chodorow & Dirolf, 2010; Banker, 2011] é um sistema de gerência de bancos de dados orientados a documentos, de alto desempenho, escalável e de código aberto. Ele é escrito em C++ e possue *drivers* para diversas linguagens de programação como C, C++, Erlang, Haskell, Java, Javascript, .NET, Python e Ruby, entre outras.

Cada instância do MongoDB pode gerenciar vários bancos de dados independentes. Cada banco de dados possui coleções que armazenam documentos. O MongoDB é livre de esquema (*schema-free*), de modo que uma coleção pode conter documentos com estruturas totalmente diferentes. As coleções do MongoDB são equiva-

[2]MongoDB: `http://www.mongodb.org/`

Tabela 2.1. Operadores condicionais do MongoDB.

Operadores	Descrição
$lt, $lte, $gt, $gte	operadores menor que, menor ou igual que, maior que, maior ou igual que, respectivamente
$in	verifica se um *array* possui algum dos valores especificados em uma lista
$nin	similar ao operador $in, porém verifica se um *array* não possui nenhum dos valores especificados em uma lista
$all	verifica se um *array* possui todos os valores especificados em uma lista
$exists	verifica se um determinado campo existe em um dado objeto
$mod	módulo - resto de uma divisão
$ne	diferente
$and, $or, $nor	
$type	verifica se um campo é de um determinado tipo BSON

lentes às tabelas de um banco de dados relacional, as quais os documentos corresponde as linhas. Os documentos são armazenados no banco de dados no formato *Binary JSON* (BSON) [BSON, 2012] que é uma serialização binária de documentos estilo JSON[3]. O BSON possui as seguintes características [BSON, 2012]: (i) leve, o formato BSON representa os dados de forma eficiente sem usar espaço extra; (ii) eficiente, em razão da utilização da representação de tipos estilo C, a codificação e decodificação dos dados de/para BSON foi projetada para ser executada rapidamente em várias linguagens.

Além dos tipos de dado básicos definidos no formato JSON (*string, number, object, array, true, false* e *null*), o MongoDB possui alguns outros tipos adicionais, sendo eles: *date, object id, binary data, regular expression* e *code*.

O MongoDB possui uma poderosa linguagem de consulta, sendo as consultas expressas como documentos JSON. O MongoDB permite a especificacão de expressões regulares em consultas e fornece uma variedade de operadores condicionais. As Tabelas 2.1 e 2.2 apresentam os operadores condicionais e exemplos de consultas JSON do MongoDB, respectivamente.

O MongoDB também possui recursos como:

- Indíces;
- Replicação assíncrona de dados entre os servidores (*replica sets* e *master-slave*);
- Arquitetura *auto-sharding* que possibilita escalabilidade horizontal;

[3]JSON: `http://www.json.org/`

Tabela 2.2. Exemplos de consultas JSON do MongoDB.

Operador	Sintaxe	Exemplo	Descrição
	{campo:valor}	{idade:27}	Seleciona os documentos que possuem o campo *idade* com valor igual a 27
$ne	{campo:{$ne:valor}}	{idade:{$ne:30}}	Seleciona os documentos que não possuem o campo *idade* com valor igual a 30. Isso inclui os documentos que não possuem esse campo
$lt	{campo:{$lt:valor}}	{idade:{$lt:30}}	Seleciona os documentos que possuem o campo *idade* com valor menor que 30
$gte	{campo:{$gte:valor}}	{idade:{$gte:5}}	Seleciona os documentos que possuem o campo *idade* com valor maior ou igual a 5
$in	{campo:{$in:[<val1>,...,<valN>]}}	{cores:{$in:["preto","branco"]}}	Seleciona os documentos que possuem o campo *cores* com valor igual a preto ou branco
$and	{$and:[{<exp1>},...,{<expN>}]}	{$and:[{v:{$gt:6}},{f:{$lte:1}}]}	Seleciona os documentos que possuem o campo v com valor maior que 6 e o campo f com valor menor ou igual a 1
$exists	{campo:{$exists:<boolean>}}	{dependentes:{$exists:true}}	Seleciona os documentos que possuem o campo *dependentes*
$regex	{campo:{$regex:<exp>}}	{nome:{$regex:"acme.*corp"}}	Seleciona os documentos que possuem o campo *nome* e cujo valor combina com a expressão regular *acme.*corp*

- *Map-reduce*;
- *GridFS*, mecanismo para armazenar grandes arquivos.
- *Shell* interativo, que permite que os usuários interajam com o MongoDB.

O MongoDB vem sendo utilizado por diversas empresas e grupos de pesquisa de variadas áreas, por exemplo, FourSquare, Bit.ly, SAP, Springer e Observatório da Web.[4]

[4]A lista atualizada de implantações de produção do MongoDB está disponível em: `http://www.mongodb.org/display/DOCS/Production+Deployments`

Capítulo 3

Watershed

Este capítulo descreve o ambiente de execução Watershed, incluindo a apresentação de sua interface de programação e a descrição detalhada dos principais pontos de sua arquitetura.

3.1 O Ambiente de Execução

O Watershed é um ambiente de execução que provê abstrações para o desenvolvimento de aplicações distribuídas que processam fluxos de dados massivos. Ele implementa o modelo de programação filtro-fluxo (*filter-stream*) [Acharya et al., 1998; Beynon et al., 2000]. Dessa forma, cada aplicação é decomposta em módulos de processamento que executam concorrentemente, explorando assim o paralelismo de tarefas. As trocas de dados entre os módulos de processamento são feitas através de canais de comunicação denominados fluxos.

No Watershed, as conexões entre os módulos de processamento são dirigidas a dados, cabendo aos desenvolvedores especificar apenas os tipos de dado que cada módulo consome e produz, não sendo necessário fazer uma explícita conexão entre eles. Com isso, é responsabilidade do ambiente realizar dinamicamente a conexão dos módulos produtores com seus respectivos consumidores.

Os módulos de processamento podem ser transparentemente replicados, ou seja, pode ser gerado um conjunto de instâncias idênticas de cada módulo. Essas instâncias podem rodar em diferentes nodos de processamento e os dados de entrada de um módulo podem ser particionados entre suas instâncias, obtendo-se assim o paralelismo de dados.

O Watershed foi projetado para executar, principalmente, aplicações que não possuem o conceito de terminação, ou seja, que executam continuamente. Ele fornece

um mecanismo de adição de novos módulos em tempo de execução. Quando necessário, os módulos das aplicações podem ser removidos do ambiente, por meio da chamada de um método de terminação no código do módulo, ou por um comando disparado do *console* do Watershed, conforme descrito na Seção 3.3.1. Esses recursos permitem que diversas aplicações executem simultaneamente no ambiente e compartilhem resultados intermediários, eliminando-se computações repetidas sobre os mesmos dados.

Para instalar o Watershed em um *cluster*, os usuários devem fornecer um arquivo XML (*Extensible Markup Language*) descrevendo a configuração do ambiente. Nesse arquivo são informados o caminho da instalação da biblioteca de comunicação de mensagens, bem como o caminho da biblioteca do Watershed. Além disso, deve ser informada a lista de máquinas a serem utilizadas. Com relação às máquinas, o Watershed permite a criação de grupos virtuais, por exemplo, pode ser definido um grupo virtual que contém apenas máquinas que possuem a API do Twitter instalada. Dessa forma, para cada máquina deve-se informar os nomes dos grupos que ela faz parte. Conforme descrito na Seção 3.2, essa informação é utilizada para determinar em qual grupo de máquinas um módulo de processamento precisa ser executado, dado o recurso necessário.

Usando as informações do documento XML de configuração, o Watershed é carregado para todos os nodos de computação disponíveis. As Figuras 3.1 e 3.2 apresentam, respectivamente, a DTD (*Document Type Definition*) utilizada para descrever formalmente e validar os arquivos XML de configuração e um exemplo desse arquivo no contexto do Watershed.

Em suma, o Watershed é um ambiente de execução que explora os recursos computacionais disponíveis, visando alcançar a máxima eficiência das aplicações que executam sobre ele, abstraindo as dificuldades inerentes ao desenvolvimento de aplicações distribuídas e paralelas. Dessa forma, o usuário precisa concentrar-se apenas no desenvolvimento dos módulos de processamento de suas aplicações. Fica a cargo do ambiente criar as instâncias dos módulos, conectar os módulos dinamicamente de acordo com uma abordagem dirigida a dados e gerenciar o roteamento das mensagens, ou seja, coordenar toda a execução que ocorre de forma paralela e distribuída.

Dois outros trabalhos estão sendo desenvolvidos por membros do grupo de pesquisa do Watershed para otimizar o uso dos recursos computacionais de modo a melhorar o desempenho das aplicações. Para tanto está sendo desenvolvido um mecanismo dinâmico de balanceamento de carga e também um mecanismo que permite a migração e elasticidade das instâncias dos módulos de processamento.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT config (global, hostdec)>
        <!ELEMENT global (ompi, server, database, processing_module)>
                <!ELEMENT ompi (#PCDATA)>
                        <!ATTLIST ompi prefix CDATA #REQUIRED>
                <!ELEMENT server (#PCDATA)>
                        <!ATTLIST server
                                  name CDATA #FIXED "ws-manager"
                                  home CDATA #REQUIRED
                                  running_dir CDATA #REQUIRED>
                <!ELEMENT database (#PCDATA)>
                        <!ATTLIST database
                                  exe_name CDATA #FIXED "ws-stream">
                <!ELEMENT processing_module (#PCDATA)>
                        <!ATTLIST processing_module
                                  exe_name CDATA #FIXED "ws-module">
        <!ELEMENT hostdec (host+)>
                <!ELEMENT host (resource+)>
                        <!ATTLIST host
                                  name CDATA #REQUIRED
                                  database_server (true|false) "true">
                        <!ELEMENT resource (#PCDATA)>
                                <!ATTLIST resource
                                          name CDATA #REQUIRED>
```

Figura 3.1. DTD do arquivo de configuração do Watershed.

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
        <global>
                <ompi prefix = "/opt/openmpi-1.5.4" />
                <server
                        home = "/home/speed/anapc/libwatershed"
                        running_dir = "/var/tmp/anapc">
                </server>
        </global>
        <hostdec>
                <host name = "hera"
                      database_server = "false">
                      <resource name = "Web" />
                </host>
                <host name = "eirene"
                      database_server = "true">
                      <resource name = "TwitterAPI" />
                      <resource name = "Web" />
                </host>
                <host name = "hefesto"
                      database_server = "true">
                      <resource name = "TwitterAPI" />
                      <resource name = "Web" />
                </host>
        </hostdec>
</config>
```

Figura 3.2. Exemplo de arquivo de configuração do Watershed.

3.2 Interface de Programação

As aplicações construídas sobre o Watershed são compostas de duas partes: (i) descrição da topologia e (ii) descrição do processamento. O usuário descreve a topologia da aplicação declarativamente, por meio de um conjunto de arquivos XML. A DTD do arquivo de descrição da topologia dos módulos de processamento é mostrada na Figura 3.3. Para cada módulo de processamento, o usuário deve fornecer um arquivo XML contendo as seguintes informações:

- **name:** nome do módulo;
- **library:** localização da biblioteca dinâmica que contém a implementação do módulo;
- **instances:** (opcional) número de instâncias do módulo;
- **arguments:** (opcional) argumentos passados para o módulo, por exemplo, um arquivo contendo dados de entrada;

Um módulo pode consumir zero ou mais tipos distintos de fluxo de entrada. Para cada fluxo de entrada são fornecidas as seguintes informações:

- **name:** nome do fluxo de entrada do módulo;
- **policy:** política de recebimento de mensagens pelas instâncias do módulo. Atualmente há três políticas implementadas, *round roubin*, *broadcast* e fluxo rotulado (*label stream*).
- **policy_function_file:** essa informação é fornecida quando a política de recebimento de mensagens selecionada é um fluxo rotulado. Com essa política quando um módulo envia uma mensagem a outro módulo, o Watershed aplica uma função *hash* para determinar a instância destino da mensagem. A *tag policy_function_file* armazena a localização da biblioteca dinâmica que contém a implementação da função *hash*.
- **output_name**: nome do fluxo de saída do módulo. Cada módulo pode produzir no máximo um tipo de fluxo de saída.
- **demand:** (opcional) conforme descrito na Seção 3.1, o Watershed faz associações virtuais de grupos de máquinas. Esses grupos são definidos no arquivo de configuração do ambiente. Por meio da *tag demand*, pode-se determinar em qual grupo de máquinas o módulo precisa ser executado, dado o recurso necessário.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT processing_module (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"
                                  policy_function_file CDATA "none">
        <!ELEMENT output (#PCDATA)>
                <!ATTLIST output
                        name CDATA #REQUIRED>
        <!ELEMENT demands (demand+)>
                <!ELEMENT demand (#PCDATA)>
                        <!ATTLIST demand
                                  name CDATA #REQUIRED>
```

Figura 3.3. DTD do arquivo XML de descrição da topologia dos módulos de processamento.

A descrição do processamento do módulo é feita procedimentalmente. O Watershed fornece uma interface de programação escrita em C++, que pode ser usada no desenvolvimento dos módulos de processamento. Os programadores devem implementar um pequeno número de métodos que são invocados pelo ambiente para tratar ações específicas, por exemplo, a chegada de novas mensagens. Entre esses métodos, os principais são:

- void Process(Message& message): processa cada mensagem recebida.
- int GetLabel(Message& message, int total_instances): invocado pelo Watershed para determinar para qual instância de um módulo deve ser enviada uma mensagem, quando a política definida para o recebimento de mensagens é fluxo rotulado.

Os seguintes métodos auxiliares podem ser utilizados pelos desenvolvedores em suas aplicações:

- string GetArgument(string argument_name): conforme descrito anteriormente, os usuários podem passar argumentos para seus módulos através de um campo no arquivo de descrição da topologia do mesmo. Esse método recupera o valor do argumento usando seu identificador.
- string GetModuleName(void): retorna o nome do módulo chamador;

- int GetNumberInstances(void): retorna o número de instâncias do módulo chamador;
- int GetRank(void): retorna o identificador da instância do módulo chamador;
- void Send(Message& message): envia uma mensagem para os consumidores do módulo chamador, respeitando a política de recebimento de mensagens de cada consumidor;
- void SynchronizeConsumers(Message& message): envia uma mensagem para todos os consumidores do módulo chamador.
- void TerminateModule(void): termina a execução do módulo chamador;

O Watershed oferece também alguns métodos que podem ser utilizados nos módulos de processamento para tratamento das mensagens.

- void* GetData(void): obtém a carga útil da mensagem;
- int GetDataSize(void): retorna o tamanho da mensagem, em bytes;
- int GetSource(void): retorna o *rank* da instância que produziu a mensagem;
- string GetSourceStream(void): retorna o nome do fluxo da mensagem. Esse método é útil para determinar de qual fluxo pertence uma mensagem, uma vez que o módulo de processamento consome mais de um fluxo de entrada;
- int GetTimestamp(void): retorna o *timestamp* de criação da mensagem;
- void SetData(void* data): atribui a carga útil da mensagem.

3.3 Arquitetura

O Watershed é composto por uma biblioteca e por *daemons* que são executados nas máquinas do *cluster* utilizado. A arquitetura do Watershed é logicamente formada por quatro componentes principais (Figura 3.4), descritos a seguir:

- Aplicativo Console: interface por meio da qual os usuários podem interagir com o ambiente, por exemplo, inserindo ou removendo módulos do Watershed.

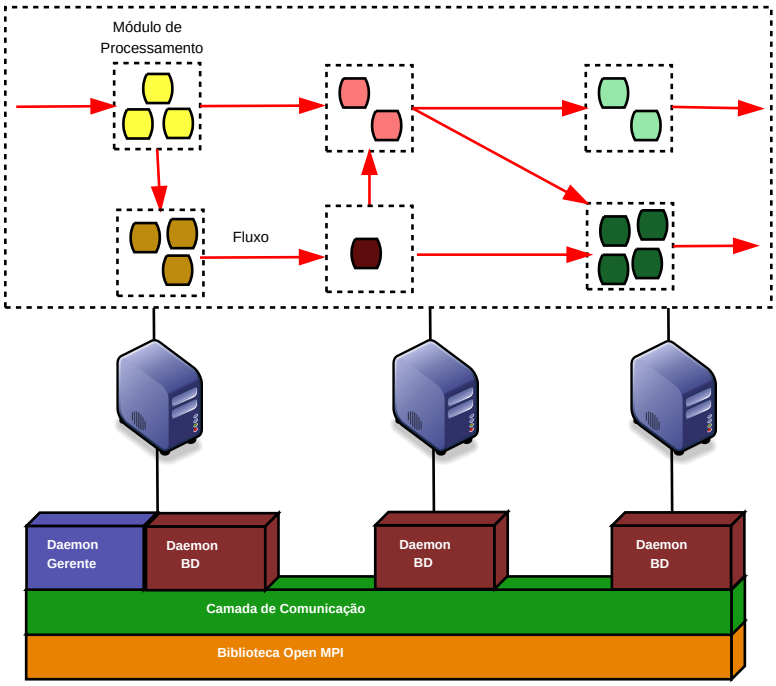


Figura 3.4. Arquitetura do Watershed

- *Daemon* gerente: componente responsável pelo controle geral da execução das aplicações. O *daemon* gerente e o aplicativo console são executados apenas na primeira máquina da lista de máquinas informada no documento XML de configuração do Watershed.

- *Daemon* de banco de dados: um subconjunto de máquinas do *cluster* executa os *daemons* de banco de dados. Esse componente é responsável por fornecer informações para realizar a conexão dinâmica dos módulos de processamento.

- Camada de comunicação: responsável pela troca de mensagens entre os módulos de processamento. A camada de comunicação foi baseada em Message-Passing Interface (MPI) e construída sobre Open MPI [Gabriel et al., 2004].

Cada um dos componentes do Watershed são descritos detalhadamente nas Seções 3.3.1 a 3.3.4.

3.3.1 Aplicativo Console

O aplicativo console é um programa por meio do qual os usuários podem interagir com o Watershed. A interação entre os usuários e o ambiente é mostrada na Figura 3.5. Os usuários podem disparar, por meio do aplicativo console, os seguintes comandos:

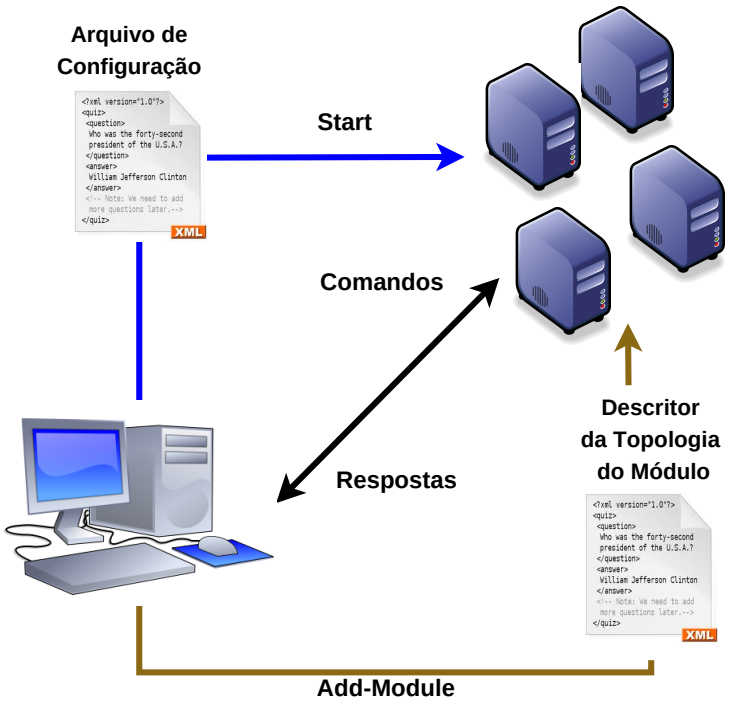


Figura 3.5. Diagrama de interação usuário/Watershed

- **start:** inicia o Watershed, isto é, o *daemon* gerente e os *daemons* de banco de dados são iniciados nas máquinas do *cluster*. O usuário deve fornecer o arquivo XML de configuração do ambiente.

- **stop:** para o ambiente Watershed que está em execução. Para tanto, todos os módulos que estão em execução são removidos e os *daemons* gerente e de banco de dados são parados.

- **status:** informa se o ambiente está em execução.

- **add-module:** adiciona módulos ao Watershed. O usuário deve fornecer como argumento o arquivo XML descritor da topologia do novo módulo.

- **remove-module:** remove do ambiente todas instâncias de um determinado módulo. O usuário deve informar o nome do módulo a ser removido.

A cada comando executado, o aplicativo console interage com o *daemon* gerente e em seguida reporta para o usuário o resultado da ação solicitada.

3.3.2 *Daemon* Gerente

O *daemon* gerente é responsável pela adição, remoção e coordenação da execução dos módulos de processamento no Watershed. Quando o usuário solicita a adição de um módulo ao ambiente (comando *add-module*), o aplicativo console envia essa solicitação para o *daemon* gerente. Utilizando as informações obtidas do arquivo descritor da topologia do módulo, o *daemon* gerente adiciona o módulo ao ambiente. As instâncias do módulo são criadas nas máquinas do *cluster*, respeitando-se a demanda de recursos do módulo e o número de instâncias requeridas. As instâncias são distribuídas entre as máquinas em *round robin*. Quando o número de instâncias não é definido pelo usuário, cada máquina que atende a demanda de recursos do novo módulo recebe uma instância do mesmo.

Quando é solicitado via console a remoção de um módulo do ambiente (comando *remove-module*), o *daemon* gerente envia uma mensagem para os módulos produtores e consumidores desse módulo informando sua remoção. Em seguida é enviada uma mensagem de terminação para todas as instâncias do módulo a ser removido. Cada instância então realiza a desconexão dos seus produtores, consumidores e dos *daemons* de banco de dados. Todas as instâncias são sincronizadas para terminarem juntas. Processo semelhante ocorre quando o usuário solicita o encerramento da execução do Watershed (comando *stop*), o *daemon* gerente envia uma mensagem para todos os módulos e *daemons* solicitando o seu encerramento.

O *daemon* gerente é responsável também por informar aos módulos as portas de conexão dos módulos produtores e consumidores, assim como as portas dos *daemons* de banco de dados.

3.3.3 *Daemon* de Banco de Dados

Os *daemons* de banco de dados são responsáveis pela manutenção das informações dos fluxos de entrada e saída dos módulos. Essas informações são utilizadas para fazer a associação dos módulos produtores com seus consumidores, e vice-versa.

Um dos objetivos do presente trabalho é desenvolver um mecanismo de persistência de fluxos de dados para o Watershed. Dessa forma, os detalhes dos *daemons* de banco de dados referentes a esse mecanismo são detalhados no Capítulo 4.

3.3.4 Camada de Comunicação

A camada de comunicação do Watershed foi baseada no padrão MPI e construída sobre Open MPI. O padrão MPI foi escolhido, pois ele apresenta algumas vantagens

em relação a outras plataformas de passagem de mensagens, além de ser um padrão maduro que provê um elevado grau de portabilidade. A camada de comunicação provê funções para realizar o envio e recebimento de mensagens, assim como funções para realizar o gerenciamento dos processos e das conexões.

No padrão MPI é utilizado um mecanismo chamado comunicador (*communicator*), que representa o domínio de comunicação dos processos. Os comunicadores se dividem em dois tipos: intracomunicador (*intracommunicator*) e intercomunicador (*intercommunicator*). A comunicação dos processos de um mesmo grupo é realizada utilizando um intracomunicador, já a comunicação entre processos de grupos distintos é feita por meio de intercomunicadores.

O aplicativo console, os daemons e as instâncias dos módulos foram implementados como processos MPI. A forma de interação desses processos é descrita a seguir.

O aplicativo console é um programa que pode ser executado a partir de uma das máquinas do *cluster* que está executando o Watershed. O aplicativo console comunica-se exclusivamente com o *dameon* gerente para enviar os comandos dos usuários. Essa comunicação segue um modelo cliente/servidor, em que o *daemon* gerente mantém uma porta aberta para receber as conexões do console. Quando o usuário dispara um comando, o aplicativo console conecta-se ao *daemon* gerente que é responsável por executar a ação solicitada. Após a conclusão do comando, o aplicativo console reporta para o usuário o resultado da ação, em seguida ele desconecta-se do *daemon* gerente e finaliza sua execução.

O *daemon* gerente forma logicamente um grupo MPI. Quando o *daemon* gerente recebe do console uma mensagem solicitando a adição de um módulo, ele envia uma requisição para o *daemon* do MPI criar as instâncias (processos) do novo módulo remotamente. Essas instâncias são agrupadas logicamente, compondo um grupo MPI.

Os *daemons* de banco de dados são criados pelo *daemon* gerente. Um grupo lógico é formado também com os *daemons* de banco de dados, esse grupo abre uma porta de conexão para os grupos de módulos de processamento.

A cada novo módulo criado é realizada a conexão de seu grupo lógico com o grupo de *daemons* de banco de dados, e é aberta uma porta para receber as conexões dos demais módulos. O grupo do novo módulo interage com o *daemon* gerente para obter os nomes e as portas de seus produtores e consumidores. Com base nessas informações são realizadas as conexões entre os módulos. Dessa forma, o novo módulo então inicia seu processamento, podendo receber e enviar mensagens. Quando é solicitada a remoção de um módulo do Watershed, o *daemon* gerente envia uma mensagem de terminação para todas as instâncias daquele módulo. Em seguida, o módulo em questão desconecta dos demais grupos (*daemons* de banco de dados, produtores e consumidores) e, por

fim, todas as instâncias do módulo em questão encerram sua execução.

A camada de comunicação garante a ordenação das mensagens enviadas entre duas instâncias, ou seja, todas as mensagens recebidas por uma instância consumidora chegarão na mesma ordem que elas foram enviadas pela instância produtora. Entretanto não é garantida nenhuma ordenação em relação as mensagens enviadas por instâncias distintas para uma instância consumidora, isto é, caso uma instância produtora envie uma mensagem no tempo t0, não necessáriamente essa mensagem vai chegar a instância consumidora antes das mensagens que foram produzidas posteriomente (tempo $>$ t0) por outras instâncias.

O Watershed implementa um mecanismo de controle de fluxos para evitar que os módulos produtores sobrecarreguem os módulos consumidores, pois, apesar do Open MPI ter um *buffer* do lado do consumidor, podem ocorrer situações que esse *buffer* cresça tanto que a memória da máquina não seja suficiente. Os detalhes do mecanismo de controle de fluxo são descritos na Seção 3.3.4.2.

3.3.4.1 Roteamento de Mensagens

Com relação ao roteamento das mensagens, deve ser definido para cada fluxo de entrada de cada módulo como os dados devem ser entregues para suas instâncias. Atualmente três políticas de recebimento de mensagens estão implementadas no Watershed:

1. *round roubin*: as mensagens de uma instância de um módulo produtor são enviadas alternadamente para as instâncias de um módulo consumidor. Toda instância produtora inicia o envio das mensagens a partir de uma instância determinada aleatoriamente. O intuito disso é tentar evitar que as instâncias produtoras enviem ao mesmo tempo mensagens para a mesma instância consumidora, sobrecarregando-a e deixando as demais ociosas.

2. *broadcast*: todas as instâncias de um módulo de processamento consumidor recebem uma cópia das mensagens destinadas a esse módulo.

3. fluxo rotulado: as mensagens de uma instância do produtor são enviadas para as instâncias de um consumidor de acordo com uma função *hash* aplicada sobre a mensagem. Essa função *hash* é fornecida pelo usuário.

3.3.4.2 Controle de Fluxo

O mecanismo de controle de fluxo de dados implementado no Watershed segue uma estratégia baseada em créditos [Liu & Panda, 2004]. No mecanismo desenvolvido cada

instância de um módulo consumidor possui um *buffer* de recebimento de mensagens de tamanho fixo. Esse *buffer* é dividido igualmente entre os produtores do módulo em questão. Para isso, cada instância consumidora envia créditos para as instâncias de seus produtores. A quantidade de créditos destinada para cada instância produtora é a razão entre o tamanho do *buffer* e o número de instâncias produtoras daquele módulo. Quando uma instância produtora envia uma mensagem, ela decrementa a quantidade de créditos referente à instância consumidora. Quando terminam os créditos da instância produtora, ela aguarda o recebimento de novos créditos.

O consumidor também armazena a quantidade de créditos de cada instância produtora. Dessa forma, a cada mensagem recebida, ele decrementa a quantidade de créditos associada à instância emissora da mensagem. Quando terminam os créditos de uma instância produtora, o consumidor recalcula a quantidade de créditos destinados a cada instância produtora e, em seguida, envia os novos créditos para essa instância produtora.

3.4 Sumário

Este capítulo descreveu em detalhes o Watershed, tendo apresentado os principais pontos de sua arquitetura e sua interface de programação. O Watershed é um ambiente de execução que provê abstrações para o desenvolvimento de aplicações distribuídas que processam fluxos de dados massivos. Entre suas principais características pode-se destacar: (i) O Watershed implementa o modelo de programação filtro-fluxo e as conexões entre os módulos de processamento são dirigidas a dados; (ii) Ele permite o desenvolvimento e execução de aplicações com topologia dinâmica, e permite que diversas aplicações executem simultaneamente no ambiente e compartilhem resultados intermediários, eliminando-se computações repetidas sobre os mesmos dados.

O próximo capítulo apresentará o mecanismo de persistência de fluxos de dados proposto para o Watershed. Esse mecanismo armazena, de forma transparente, os dados produzidos pelos módulos de processamento e possibilita o consumo tanto de dados históricos quanto de atuais.

Capítulo 4

Persistência de Fluxos de Dados

Esse capítulo descreve o mecanismo de persistência proposto, são apresentados os requisitos levantados, a interface de programação desenvolvida, os principais pontos da arquitetura e os detalhes de implementação.

4.1 Análise de Requisitos

A seguir são apresentados os requisitos funcionais do mecanismo de persistência com suas respectivas motivações:

- Armazenar os dados produzidos pelos módulos: para que os módulos de processamento possam consumir dados atuais e históricos, o mecanismo de persistência deve ser capaz de armazenar os dados produzidos pelos módulos de processamento;
- Permitir o armazenamento e manipulação de dados semiestruturados: como o Watershed é um ambiente genérico, as aplicações que executam nele podem lidar com dados semiestruturados, assim o mecanismo de persistência deve possibilitar o armazenamento e manipulação de dados semiestruturados.
- Filtrar dados por data: para aumentar a flexibilidade do Watershed, o mecanismo de persistência deve manter a data de persistência dos dados e possibilitar que os usuários determinem para cada módulo, a data inicial dos dados históricos que são de interesse.
- Possibilitar o consumo de dados atuais e históricos: O mecanismo de persistência deve possibilitar que tanto dados atuais como dados históricos possam ser consumidos pelos módulos de processamento;

- Filtrar dados de um fluxo: visando aumentar a flexibilidade do Watershed, o mecanismo de persistência deve possibilitar a criação e execução de consultas personalizadas para filtrar de um fluxo específico apenas os dados de interesse.

- Exibir informações dos dados produzidos: para que os usuários possam ter conhecimento dos dados disponíveis para consumo, o mecanismo de persistência deve, sob demanda, exibir para o usuário informações sobre os tipos, quantidades e distribuição dos dados atuais e armazenados.

4.2 Visão Geral do Mecanismo de Persistência

De forma transparente para o usuário, todos os dados produzidos pelos módulos de processamento são persistidos em bancos de dados. O atual projeto do mecanismo de persistência não trata do problema de remoção automática dos dados.

Os usuários devem determinar para cada fluxo de entrada dos módulos se é desejado o consumo de dados históricos. Nos casos em que esse consumo é desejado, os usuários podem definir a data inicial de criação dos dados de interesse, através do documento XML de descrição da topologia do módulo. Assim, apenas dados que foram inseridos nos bancos de dados com data maior ou igual à data inicial informada são enviados aos módulos.

Um importante recurso provido pelo mecanismo de persistência é o apoio a criação e execução de filtros sobre os dados. Os usuários podem definir consultas que são utilizadas para selecionar de um fluxo de dados específico, apenas os dados de interesse. Por exemplo, dado um módulo que consome dados do fluxo de notícias é possível criar um filtro para consumir apenas as notícias relacionadas a Dilma, ou seja, que possuem a palavra Dilma. O formato de representação dessas consultas será descrito na Seção 4.6

4.3 Interface de Programação

O mecanismo de persistência proposto estende a interface de programação do Watershed apresentada no Capítulo 3.

Os seguintes métodos de tratamento de mensagens foram modificados para lidar apenas com dados no formato JSON. A razão dessa mudança será explicada na Seção 4.6.

- void SetData(Json* data): atribui a carga útil da mensagem.

- Json* GetData(void): obtém a carga útil da mensagem;

No arquivo de descrição da topologia dos módulos foram inseridas informações de configuração dos fluxos de entrada e saída dos módulos de processamento:

- **reader_persisted_data**: informa se o módulo consome dados persistidos.
- **query**: (opcional) consulta utilizada para filtrar os fluxos de entrada de um módulo.
- **initial_date**: (opcional) data inicial de criação dos dados persistidos de interesse (formato yyyy-mm-dd hh:mm:ss).
- **structure_file**: os usuários devem informar a estrutura dos fluxos de saída de cada módulo de processamento. Isso deve ser feito por meio de um documento JSON. A *tag* structure_file armazena a localização desse documento.

A informação da estrutura dos fluxos de saída dos módulos é importante, pois com ela os desenvolvedores podem avaliar se algum fluxo existente no ambiente pode ser consumido por novos módulos criados.

Para que os usuários possam ter conhecimento dos tipos de fluxo de dados que estão sendo produzidos pelos módulos, o Watershed fornece dois comandos que podem ser executados por meio do aplicativo console. As Figuras 4.1 e 4.2 mostram exemplos das saídas geradas por esses comandos.

- **list-active-streams:** lista todos os tipos de fluxos de dados que estão sendo produzidos por módulos que estão ativos no ambiente.
- **list-dist-streams:** lista os tipos de fluxos de dados persistidos, informando a distribuição dos dados entre as instâncias de banco de dados existentes.

```
[2012-08-20 15:16:29] INFO - listing active streams

Stream: Texto_Sem_StopWords
Structure: /home/ProcessamentoTweets/textoSemStopWords.txt

Stream: Numeros
Structure: /home/ProdutorConsumidor/numeros.txt
```

Figura 4.1. Exemplo da saída gerada pelo comando *list-active-streams*.

```
[2012-08-20 15:23:22] INFO - listing distribution of streams
Host: crystal8
        Stream: Texto_Tweet             amount: 1000000
        Stream: Texto_Sem_StopWords     amount: 2000000

Host: crystal1
        Stream: Numeros                 amount: 500
        Stream: Texto_Sem_StopWords     amount: 1000000
```

Figura 4.2. Exemplo da saída gerada pelo comando *list-dist-streams*.

Vale ressaltar que nesta implementação, assume-se que os usuários são colaboradores e não-maliciosos, uma vez que dados de fluxos diferentes são visíveis a todos os usuários, porém como trabalho futuro pretende-se incluir mecanismos de segurança, como Lista de Controle de Acesso [Flynn, 2002].

4.4 Arquitetura do Mecanismo de Persistência

Para atender aos requisitos funcionais definidos na Seção 4.1 e permitir o armazenamento de fluxos de dados no Watershed, o mecanismo de persistência deve: (i) lidar com o armazenamento de grandes volumes de dados; (ii) permitir processamento distribuído; (iii) garantir a entrega dos dados, atuais e históricos, aos módulos sem perda ou duplicação; (iv) utilizar um formato de representação de dados (de entrada e saída) que permita dados semiestruturados e seja compacto, para economizar espaço de armazenamento e reduzir consumo de banda de rede durante o intercâmbio dos dados.

Em um modelo centralizado, a sobrecarga do banco de dados pode gerar uma degradação do sistema, uma vez que ele torna-se incapaz de responder às requisições do sistema de forma escalável. O Watershed é um ambiente de processamento paralelo e distribuído, sendo desnecessário nesse cenário a centralização dos dados. Para evitar uma possível degradação do desempenho do Watershed devido à sobrecarga do banco de dados, definiu-se que todos os computadores do *cluster* no qual o Watershed está executando devem possuir um banco de dados. No projeto atual cada instância de módulo de processamento escreve seus dados de saída apenas no banco de dados local a ela, entretanto algum outro critério pode vir a ser utilizado para otimizar a escolha da instância do banco de dados que cada módulo de processamento deve escrever.

O funcionamento geral do mecanismo de persistência de fluxos de dados é mostrado na Figura 4.3 e descrito a seguir.

De forma transparente para o usuário, todos os dados produzidos pelos módulos

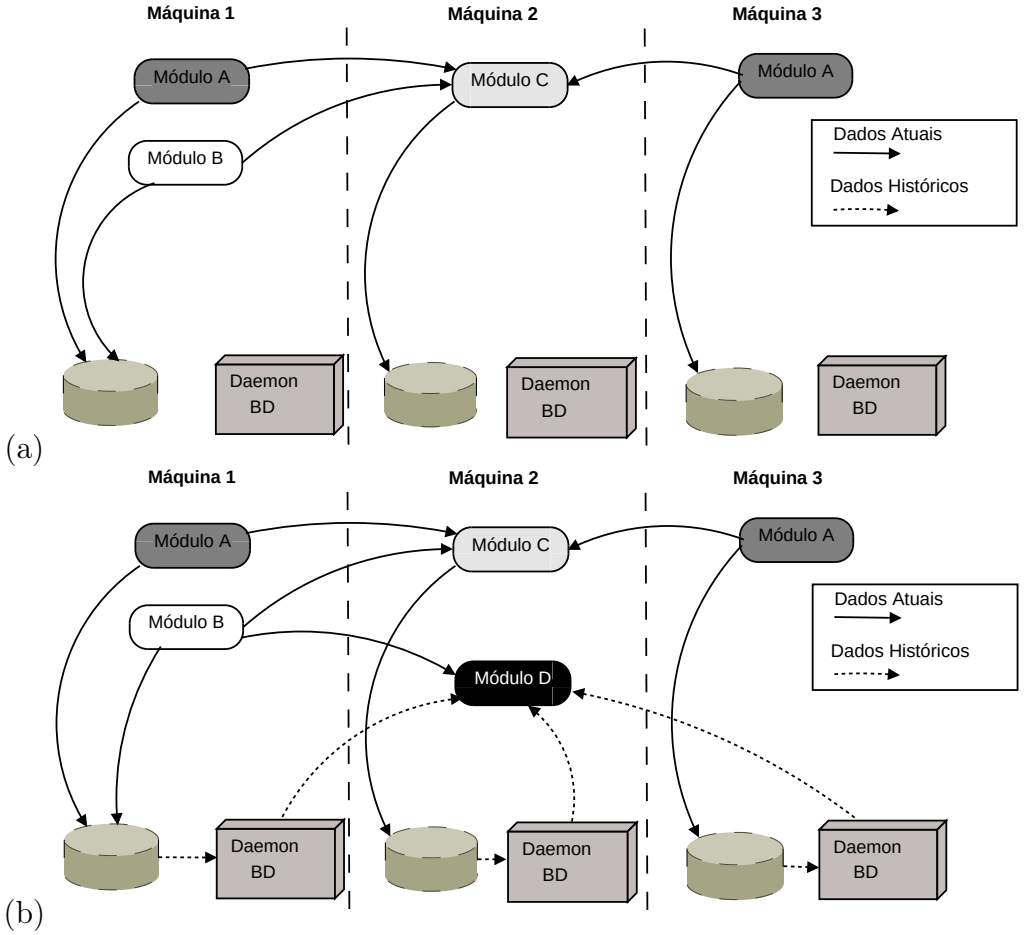


Figura 4.3. Funcionamento do mecanismo de persistência do Watershed. a) Os dados produzidos pelos módulos de processamento são persistidos e depois enviados para os módulos consumidores. b) Quando um novo módulo é inserido no Watershed, ele recebe os dados históricos dos *daemons* de banco de dados.

de processamento são persistidos em bancos de dados e depois enviados diretamente dos módulos produtores para os módulos consumidores. Quando um módulo é adicionado ao Watershed, os *daemons* de banco de dados recebem as informações da topologia desse módulo, em seguida eles recuperam de forma assíncrona os dados armazenados que são de interesse do novo módulo. A seleção desses dados armazenados é feita da seguinte forma: inicialmente é verificado para cada tipo de fluxo de entrada do novo módulo se foi fornecida alguma consulta para filtragem dos dados. Em caso positivo, essa consulta é executada, de modo que somente são selecionados os dados cuja consulta teve retorno. De forma semelhante, outro filtro é executado para selecionar apenas os dados que tenham sido criados após a data de criação informada no arquivo XML de descrição da topologia do módulo. Caso nenhuma data tenha sido informada todos os dados são retornados. Por fim, cada *daemon* de banco de dados envia os dados

selecionados para o novo módulo, respeitando a política de recebimento de mensagens definida para cada módulo. Os dados armazenados são enviadas pelos *daemons* de banco de dados sem ser estabelecida nenhuma ordenação específica.

Os componentes do Watershed relacionados a persistência de fluxos de dados são: *daemons* de banco de dados, camada de comunicação e módulos de processamento.

Os ***daemons* de banco de dados** selecionam e enviam, para cada novo módulo adicionado ao Watershed, os dados históricos que sejam de seu interesse. Em outras palavras, os módulos que foram configurados para consumir, tanto dados atuais quanto dados históricos, receberão dos *daemons* de banco de dados todos os dados armazenados cujo tipo seja igual a algum dos seus tipos de entrada e que atendam os filtros definidos pelos usuários. No Watershed com mecanismo de persistência todos os computadores do *cluster* precisam ter um *daemon* de banco de dados.

Os **módulos de processamento** são responsáveis por: (i) persistir todos seus dados de saída; (ii) executar as consultas definidas pelos usuários para filtrar do seu fluxo de saída os dados que são de interesse de cada módulo consumidor.

A **camada de comunicação** deve realizar o controle de fluxo das mensagens de dados enviadas dos *daemons* de banco de dados para os módulos de processamento.

Os **bancos de dados** que estão distribuídos nos computadores do *cluster* armazenam os fluxos de dados.

4.5 Adição Dinâmica de Módulos de Processamento

O envio dos dados para os módulos de processamento deve ser realizado sem perda ou duplicação. Essa é uma tarefa complexa, visto que, um mesmo dado pode ser enviado para diferentes módulos em distintos períodos de tempo (caso dos dados armazenados). Durante o projeto do mecanismo de persistência foi elaborada uma estratégia para lidar com essa tarefa. Foi definido que os dados atuais devem ser enviados diretamente dos módulos produtores para os módulos consumidores. Já os dados armazenados devem ser enviados pelos *daemons* de banco de dados para os novos módulos logo após serem adicionados ao Watershed.

No mesmo momento em que os *daemons* estão selecionando os dados armazenados que devem ser enviados para um novo módulo, também estão sendo produzidos e inseridos novos dados nos bancos de dados. Logo foi preciso estabelecer uma forma para determinar com precisão o limite entre os dados que são considerados históricos e devem ser enviados pelos *daemons*, e os que são considerados atuais e devem ser

enviados pelos módulos produtores. A solução adotada é apresentada nos Algoritmos 1 e 2.

Algoritmo 1: Módulo de Processamento

Entrada: N: Número de instâncias do daemon de banco de dados
P: Grupo dos módulos produtores do novo módulo de processamento
D: Grupo de daemons de banco de dados
C: quantidade de créditos

```
1  início
2      ConectaDaemonBD();
3      para cada instancia d ∈ D faça
4          EnviaCreditoParaDaemonBD(C, d);
5      fim para cada
6      ConectaProdutores();
7      Sincroniza();
8      para cada instancia d ∈ D faça
9          mensagemSaida.dados ← ObtemNomeModulo();
10         mensagemSaida.codigoMensagem ← 1;
11         EnviaMensagemParaDaemonBD(mensagemSaida, d);
12     fim para cada
13     numeroMensagens ← 0;
14     enquanto numeroMensagens < N faça
15         RecebeMensagemDoDaemonBD(mensagemEntrada);
16         se mensagemEntrada.codigoMensagem = 2 então
17             numeroMensagens ← numeroMensagens +1;
18         fim se
19     fim enqto
20     Sincroniza();
21     para cada instancia p ∈ P faça
22         EnviaCreditoParaProdutor(C, p);
23     fim para cada
24 fim
```

Quando um módulo é adicionado ao Watershed cada uma de suas instâncias se conecta com os *daemons* de banco (Algoritmo 1, linha 2) e com os seus módulos produtores (Algoritmo 1, linha 6). Quando todas as conexões são estabelecidas os *daemons* de banco de dados são avisados (Algoritmo 1, linhas 8-12). Em seguida os *daemons* de banco de dados verificam quais dados armazenados devem ser enviados para as instâncias do novo módulo de processamento (Algoritmo 2, linhas 4-9). Quando essa verificação é finalizada os *daemons* de banco de dados enviam uma mensagem de aviso para as instâncias do novo módulo (Algoritmo 2, linha 12). Somente depois disso, o novo módulo envia para todos os seus produtores os créditos que são utilizados

Algoritmo 2: Daemon de Banco de Dados

```
   Entrada: N: novo módulo de processamento
 1 início
 2 |  RecebeMensagemDoModulo(mensagemEntrada);
 3 |  nomeModulo ← mensagemEntrada.dados;
 4 |  listaFluxosEntrada ← ObtemFluxosEntrada(nomeModulo);
 5 |  para cada fluxo de dados de entrada f ∈ listaFluxosEntrada faça
 6 |  |  se f.LeDadosPersistentes = verdadeiro então
 7 |  |  |  listaDados.adiciona(LeBD(f.nome, f.dataInicial, f.consulta));
 8 |  |  fim se
 9 |  fim para cada
10 |  destino ← mensagemEntrada.fonte;
11 |  mensagemSaida.codigoMensagem ← 2;
12 |  EnviaMensagemParaModulo(mensagemSaida, destino);
13 |  para cada dado d ∈ listaDados faça
14 |  |  mensagemSaida.dados ← d;
15 |  |  AtualizaCreditos(nomeModulo, f.politica);
16 |  |  se f.politica = broadcast então
17 |  |  |  para cada instancia i ∈ N faça
18 |  |  |  |  EnviaMensagemParaModulo(mensagemSaida, i);
19 |  |  |  fim para cada
20 |  |  fim se
21 |  |  senão
22 |  |  |  se f.politica = round robin então
23 |  |  |  |  destino ← ObtemProximoDestino();
24 |  |  |  fim se
25 |  |  |  senão
26 |  |  |  |  destino ← ObtemRotulo(mensagemSaida);
27 |  |  |  fim se
28 |  |  |  EnviaMensagemParaModulo(mensagemSaida, destino);
29 |  |  fim se
30 |  fim para cada
31 fim
```

no controle de fluxo (Algoritmo 1, linhas 21-23). Caso algum módulo produtor tente enviar dados para o novo módulo no período entre a conexão e a verificação dos *daemons* de banco de dados, ele ficará aguardando (paralizado) até que seja recebida a mensagem com os primeiros créditos enviados pelo novo módulo. Vale ressaltar que apenas os módulos produtores do novo módulo podem vir a ficar momentaneamente paralizados.

Conforme apresentado no Algoritmo 2, linha 15, os *daemons* de banco de dados utilizam controle de fluxo para enviarem os dados históricos para os módulos. Em virtude disso, o mecanismo de controle de fluxo descrito na Subseção 3.3.4.2 foi adaptado. Assim, quando um módulo é adicionado ao ambiente, ele divide seus créditos entre todas as instâncias dos seus produtores e dos *daemons* de banco de dados. Após o envio de todos os dados armazenados de interesse para um novo módulo, cada *daemon* de banco de dados envia uma mensagem para as instâncias desse módulo informando o término do envio dos dados persistidos. Quando essas mensagens são recebidas, cada instância do novo módulo retira o *daemon* emissor da mensagem da partilha de créditos. Esse processo é repetido para todas as instâncias dos *daemons* de banco de dados, até que os créditos do novo módulo sejam partilhados apenas entre os módulos produtores.

4.6 Detalhes de Implementação

Com relação à representação dos dados de entrada e saída dos módulos de processamento, optou-se por utilizar o formato JSON. Isso significa que a carga útil das mensagens enviadas e consequentemente recebidas pelos módulos de processados devem ser documentos JSON. Foram adicionados métodos à interface de programação do Watershed para facilitar a criação e manipulação dos documentos JSON nas aplicações dos usuários. Para criação dos documentos JSON estão disponíveis métodos que possibilitam a adição de campos, cujos tipos podem ser *int, float, double, boolean, string, date, array, null* e *JSON*. Para manipulação dos documentos JSON estão disponíveis métodos para obtenção de campos específicos. Além disso, o usuário pode utilizar, quando necessário, métodos de conversão de *string* para o formato JSON e vice-versa.

As vantagens do formato JSON incluem sintaxe simples, aprendizado fácil e o menor tamanho dos arquivos gerados quando comparado a outros como XML. O tamanho dos arquivos é um aspecto importante, uma vez que os dados são transmitidos pela rede.

Para armazenar os dados optou-se por utilizar o sistema MongoDB. Esse sistema foi escolhido pois atende aos requisitos necessários para o desenvolvimento do meca-

nismo de persistência. Dentre esses requisitos podem ser destacados alto desempenho, suporte a consultas, armazenamento de documentos estilo JSON e acesso por meio de um *driver* C++.

Conforme descrito anteriormente, os desenvolvedores podem especificar consultas para selecionar de um fluxo de dados específico apenas os dados de interesse. Essas consultas devem ser expressas utilizando a linguagem de consultas do MongoDB. A linguagem de consultas do MongoDB está descrita na Seção 2.3.

A Figura 4.4 apresenta um exemplo que ilustra como os fluxos de dados do Watershed foram modelados no MongoDB. Cada máquina do *cluster* possui uma instância do MongoDB. Os fluxos de dados foram modelados como coleções e os dados foram modelados como documentos. No exemplo o módulo de processamento *LeitorJornal* possui duas instâncias e produz o fluxo de saída *Notícias*. Quando esse módulo foi adicionado ao ambiente o *daemon* de banco de dados criou duas coleções *Notícias*, uma no MongoDB da máquina 1 e outra na máquina 2. Nessas coleções são armazenados todos os dados (documentos) produzidos por essas instâncias. O Módulo *LeitorRevista* possui apenas uma instância e também produz o fluxo de dados *Notícias*, assim seus dados de saída (documentos) são inseridos na coleção *Notícias* já existente (máquina 1). Conforme descrito anteriormente, as instâncias dos módulos de processamento escrevem seus dados de saída no MongoBD da sua máquina local.

Cada *daemon* de banco de dados mantém uma lista com os tipos de fluxos de dados, isto é coleções, que estão armazenados no seu MongoDB local. Os *daemons* de banco de dados são responsáveis por enviar os dados históricos para os módulos de processamento adicionados ao Watershed, que tenham esse interesse. Para tanto, cada *daemon* de banco de dados consulta em sua lista se há coleções no seu MongoDB local que correspondem aos tipos de fluxos de entrada do novo módulo. Em caso positivo os *daemons* de banco de dados selecionam das coleções os documentos que atendem aos filtros que os usuários definiram para cada fluxo de entrada. Essa seleção é feita utilizando recursos de consulta do MongoDB.

4.7 Sumário

Este capítulo apresentou o mecanismo de persistência de fluxos de dados projetado para o Watershed. As principais características do mecanismo incluem armazenamento e manipulação de dados semiestruturados, processamento distribuído e também facilidades para criação e execução de filtros sobre os dados. Foram descritos os requisitos do mecanismo, a interface de programação desenvolvida, os principais aspectos

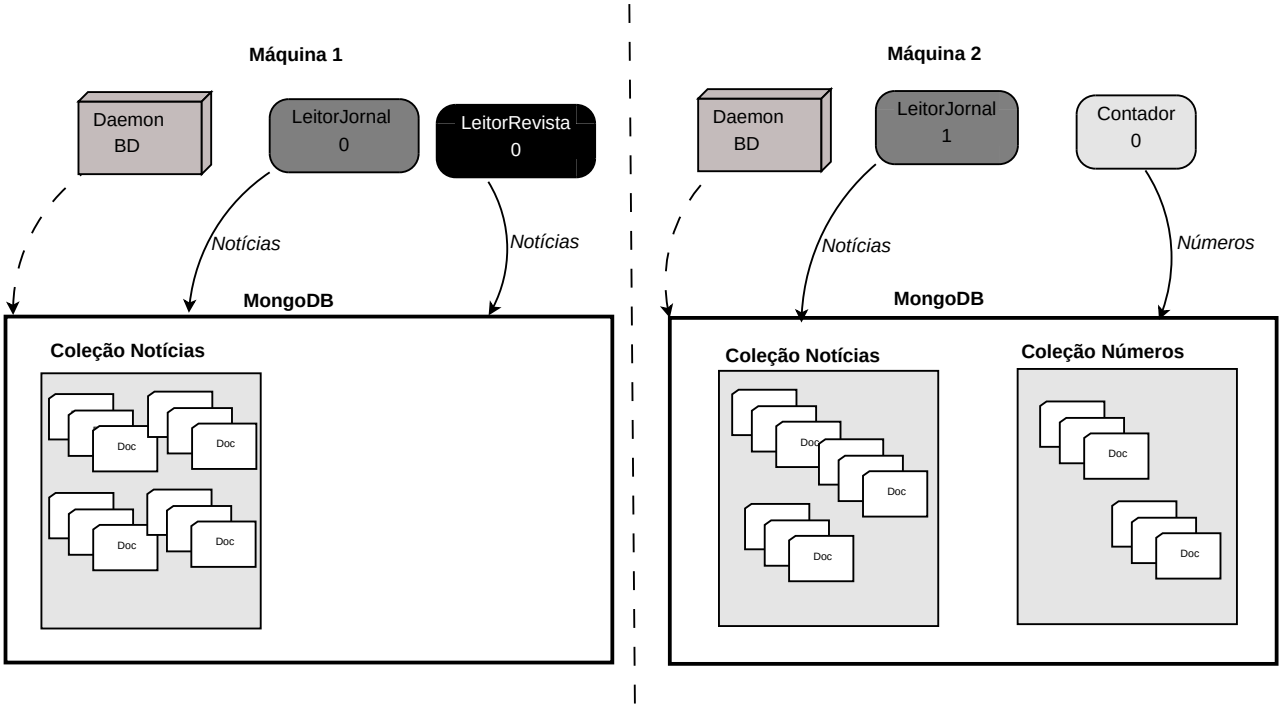


Figura 4.4. Exemplo da modelagem dos fluxos de dados no MongoDB

da arquitetura e detalhes de implementação.

Para ilustrar o uso do Watershed, o Apêndice A apresentará a descrição detalhada do desenvolvimento de uma aplicação nesse ambiente.

O próximo capítulo apresentará a avaliação experimental realizada. Serão descritas as aplicações utilizadas e a configuração dos experimentos. Por fim, serão mostrados e discutidos os resultados obtidos.

Capítulo 5

Avaliação Experimental

Este capítulo apresenta a avaliação experimental realizada com o mecanismo de persistência de fluxos de dados proposto nesta dissertação. O restante do capítulo está dividido da seguinte forma. A Seção 5.1 descreve as aplicações utilizadas na avaliação experimental. Em seguida, a Seção 5.2 descreve a configuração dos experimentos. Por fim, a Seção 5.3 apresenta e discute os resultados obtidos.

5.1 Aplicações

A avaliação experimental foi realizada utilizando três aplicações: `Processamento de` *`Tweets`*, `Detecção de Congestionamentos` e `Produtor-Consumidor`. As duas primeiras aplicações foram escolhidas pois o Watershed foi projetado principalmente para executar aplicações com essas características, ou seja, aplicações que processam de forma contínua fluxos de dados, que no caso foram *tweets* e dados de sensores de tráfego. Para facilitar a realização de algumas análises mais refinadas foi utilizada uma aplicação fictícia e genérica, denominada `Produtor-Consumidor`, uma vez que ela é mais flexível e adaptável.

5.1.1 Processamento de Tweets

A aplicação `Processamento de` *`Tweets`* realiza a remoção de *stopwords* e radicalização de palavras de *tweets*. As etapas de remoção de *stopwords* e radicalização de palavras são muito utilizadas em diversas aplicações de análise que processam grandes coleções de dados vindos da Web e inferem informações relevantes

No Watershed, a aplicação `Processamento de` *`Tweets`* foi decomposta em três módulos de processamento, como pode ser visto na Figura 5.1. A seguir cada um desses

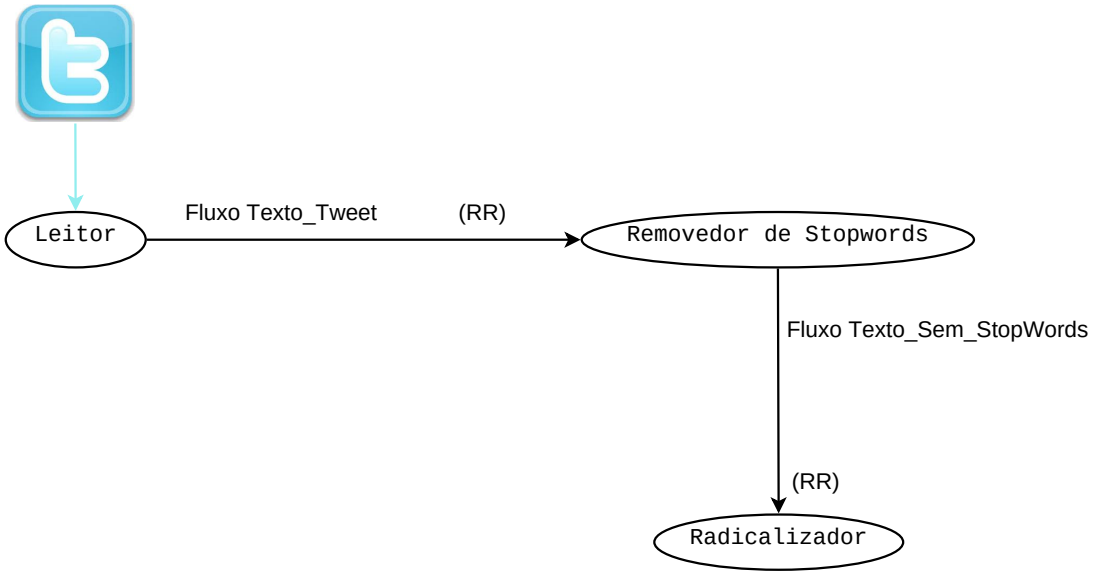


Figura 5.1. Módulos de processamento da aplicação Processamento de Tweets.

módulos é descrito de forma sucinta:

- `Leitor` - É o módulo responsável por realizar a leitura dos *tweets* que foram previamente coletados e armazenados em arquivos texto. Para cada *tweet* lido é realizada a extração de seu texto, seguida da remoção deste texto de todos os *links* e citações. O módulo produz como saída o texto remanescente de cada *tweet*.

- `Removedor de Stopwords` - Esse módulo consome textos. Ele é responsável por remover as *stopwords* presentes em cada texto de entrada. A política de recebimento de mensagens adotada é *round robin*.

- `Radicalizador` - É o módulo responsável por reduzir todas as palavras do texto de entrada para o seu radical. A política de recebimento de mensagens adotada é *round robin*.

Na aplicação `Processamento de Tweets`, obtou-se por utilizar *tweets* previamente coletados ao invés de realizar a coleta em tempo real, pois com a leitura dos dados de arquivos é possível controlar melhor os experimentos e evitar que o tempo de obtenção dos dados interfira nos resultados.

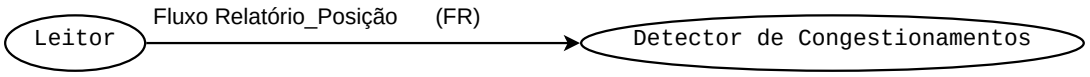


Figura 5.2. Módulos de processamento da aplicação Detecção de Congestionamentos.

5.1.2 Detecção de Congestionamentos

Essa aplicação foi desenvolvida para simular o monitoramento do tráfego de vias expressas com o objetivo de detectar congestionamentos. A partir de informações enviadas por veículos que estão trafegando em vias expressas, a aplicação continuamente calcula a velocidade média dos carros em cada via. Quando a velocidade média está abaixo de um limiar, possivelmente devido a um congestionamento, a aplicação emite um aviso.

O simulador de tráfego MITSIM[1] foi utilizado para gerar as informações do trânsito nas vias expressas. Esse simulador gera um conjunto de veículos que transmitem a cada 30 segundos um relatório de posição, contendo informações como tempo da simulação, identificador do veículo, velocidade do veículo em MPH, identificador da via expressa, segmento e direção da via expressa.

A aplicação `Detecção de Congestionamentos` foi implementada no Watershed por meio de dois módulos de processamento (Figura 5.2):

- `Leitor` - É o módulo responsável por realizar a leitura dos relatórios de posição de veículos. Esses dados foram previamente gerados pelo simulador e armazenados em arquivos texto. O módulo Leitor produz o fluxo de saída denominado *Relatório_Posição*;

- `Detector de Congestionamentos` - Esse módulo consome dados do fluxo *Relatório_Posição*. A cada minuto é calculada a velocidade média dos veículos em cada segmento de via. Caso a velocidade média de algum segmento seja inferior a um limiar preestabelecido, um aviso é emitido. A política de recebimento de mensagens adotada pelo módulo `Detector de Congestionamentos` é a de fluxo rotulado. Os relatórios de posição enviados pelo módulo `Leitor` são particionados para as instâncias do módulo `Detector de Congestionamentos` de acordo com o identificador do segmento da via expressa, de modo que cada instância fica responsável por um determinado conjunto de segmentos.

[1]MITSIM: `http://mit.edu/its/mitsimlab.html`

5.1.3 Produtor-Consumidor

A aplicação fictícia `Produtor-Consumidor` é composta por de módulos de processamento: `Produtor` e `Consumidor`. O módulo `Produtor` realiza o cálculo de algumas funções trigonométricas e produz o fluxo de saída denominado *Números*. Cada dado do fluxo *Números* contém quatro números inteiros. O módulo `Consumidor` não possui fluxo de saída, ele apenas consome dados do fluxo *Números* e realiza o cálculo de funções trigonométricas. A política de recebimento de mensagens adotada pelo módulo `Consumidor` é a *round robin*.

5.2 Configuração dos Experimentos

Os experimentos realizados neste trabalho foram executados em um *cluster* composto de doze computadores interligados por uma rede *Ethernet* de 1 GB. Cada computador é equipado com um processador Intel Core 2 CPU 6420 com *clock* de 2.13GHz, 2GBytes de memória RAM e sistema operacional Linux.

Em todos os experimentos foi inserida apenas uma instância de um módulo de processamento por computador, para que a concorrência por recursos não tivesse impacto nos resultados. Todos os tempos apresentados na avaliação experimental correspondem ao valor médio obtido de quatro execuções.

Os resultados apresentados nas Seções 5.3.3 a 5.3.5 correspondem a experimentos realizados apenas com a aplicação `Produtor-Consumidor`, pois sendo adaptável, ela viabilizou a análise de certas características do mecanismo de persistência.

5.3 Resultados

5.3.1 Custo da Utilização do Mecanismo de Persistência

Uma série de experimentos foram realizados com o objetivo de avaliar o impacto do mecanismo de persistência proposto no desempenho das aplicações. Para tanto foi gerada uma versão do Watershed sem o mecanismo de persistência, por meio do uso de diretivas de compilação condicional[2]. Inicialmente foram executados experimentos para avaliar o custo adicionado ao tempo de execução das aplicações para persistir os dados dos seus fluxos de saída. Nesses experimentos todos os módulos de processamento das aplicações foram inseridos e executados simultaneamente no Watershed. Dessa forma,

[2]As diretivas de compilação condicional permitem incluir ou descartar partes do código de um programa, se uma determinada condição é satisfeita

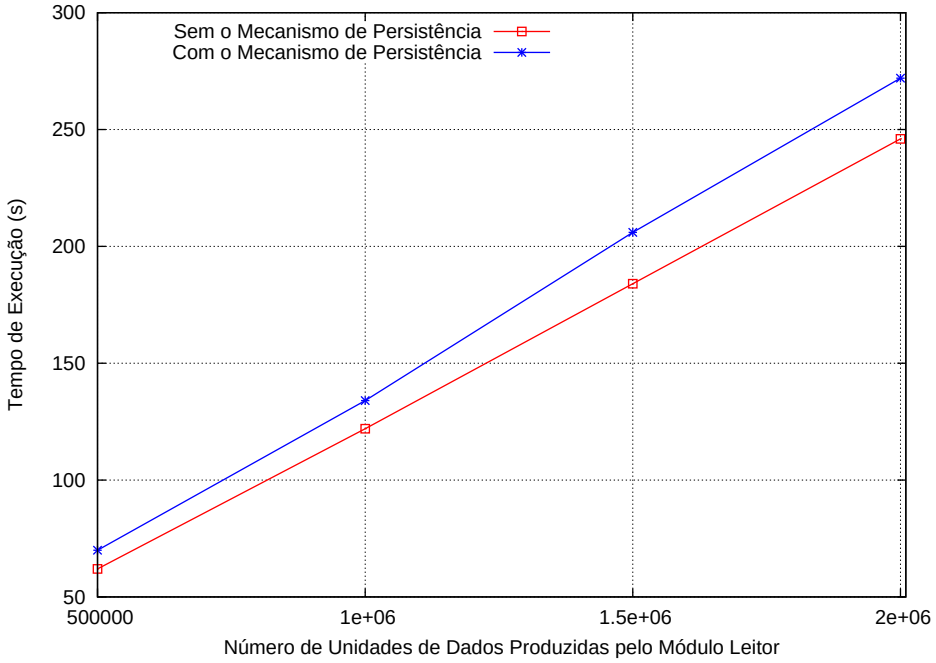


Figura 5.3. Custo de utilização do mecanismo de persistência na aplicação Detecção de Congestionamentos.

todos os dados produzidos pelos módulos foram persistidos e em seguida enviados diretamente para os módulos consumidores, não sendo lido nenhum dado armazenado. Em todas as aplicações de teste foram utilizadas três instâncias para cada módulo de processamento. Por exemplo, para a aplicação `Processamento de` *`Tweets`* foram utilizadas três instâncias do módulo `Leitor`, três instâncias do módulo `Removedor de Stopwords` e três instâncias do módulo `Radicalizador`. O tempo de execução apresentado para cada aplicação corresponde ao maior tempo obtido pelas instâncias que compõem a mesma.

Nos experimentos realizados utilizando a aplicação `Detecção de Congestionamentos`, o número de dados produzidos por cada instância do módulo `Leitor` foi variado de 500.000 a 2.000.000. A Figura 5.3 apresenta os tempos de execução obtidos utilizando o Watershed com e sem o mecânismo de persistência. Pode-se observar que em ambas as versões do Watershed o aumento no tempo de execução da aplicação é proporcional ao aumento do número de dados produzidos. A diferença entre os tempos obtidos utilizando o Watershed com e sem o mecanismo de persistência foi de no máximo 12%.

Nos experimentos com a aplicação `Processamento de` *`Tweets`* o número de dados produzidas por cada instância do módulo `Leitor` também foi variado de 500.000 a 2.000.000. Conforme pode ser notado na Figura 5.4, com as ambas as versões do

Watershed, os tempos obtidos também foram proporcionais à quantidade de dados processados. A diferença entre os tempos obtidos com e sem o mecanismo de persistência foi de no máximo 13%.

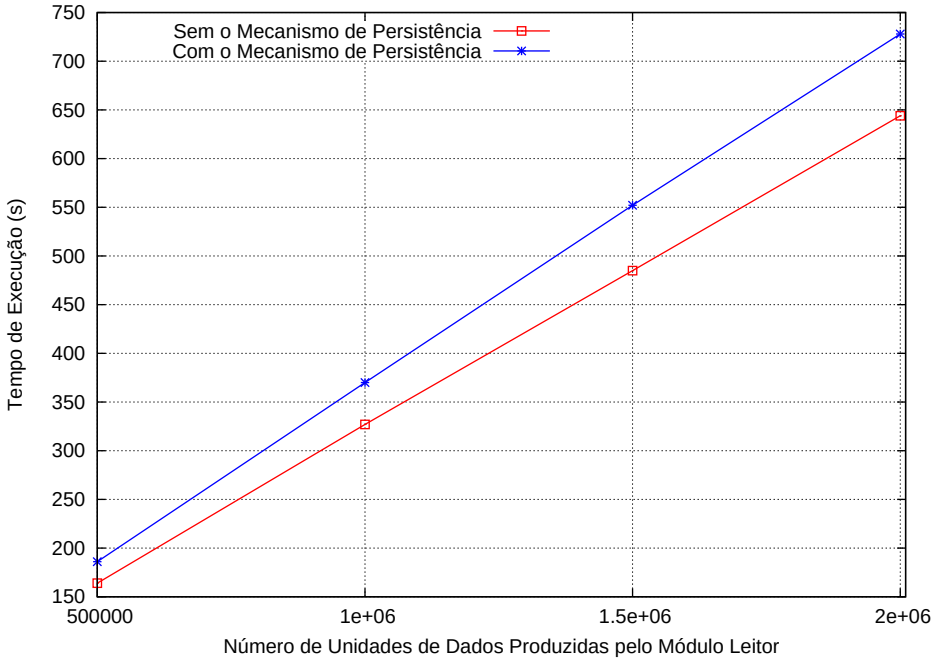


Figura 5.4. Custo da utilização do mecanismo de persistência na aplicação Processamento de Tweets.

Nos experimentos com a aplicação `Produtor-Consumidor`, o número de dados de saída produzidos por cada instância do módulo `Produtor` foi variado de 600.000 a 2.400.000. Os tempos de execução obtidos utilizando as versões do Watershed com e sem o mecanismo de persistência são apresentados na Figura 5.5. Os resultados mostram que a diferença entre os tempos obtidos com as duas versões foi de no máximo 13%.

5.3.2 Escalabilidade

Para avaliar a escalabilidade do Watershed, com e sem o mecanismo de persistência, foram realizados experimentos que mediram os tempos de execução das aplicações de teste, em face à variação do número de instâncias dos módulos de processamento. Em cada configuração foi utilizado o mesmo número de instâncias para os módulos de uma mesma aplicação. O número de instâncias foi variado de um a quatro, com uma instância por computador. Foram utilizados 1.000.000 de *tweets*, 4.000.000 relatórios de posição e 4.000.000 dados do fluxo *Números* nos experimentos realizados

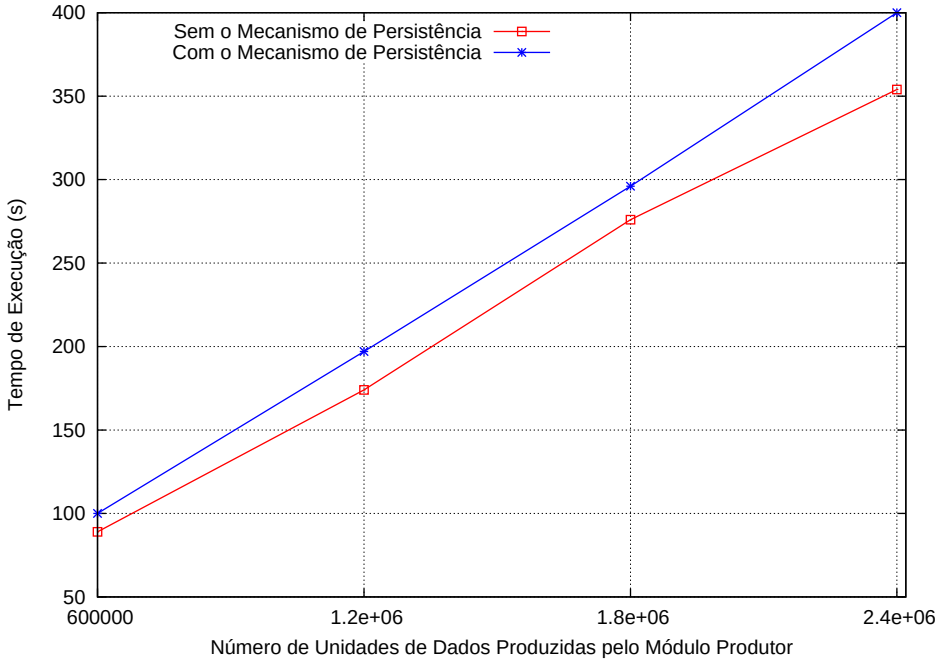


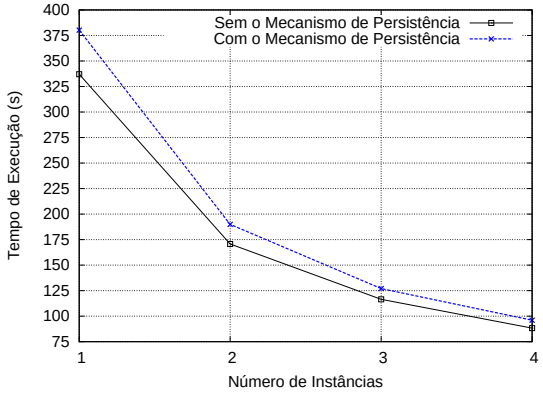
Figura 5.5. Custo da utilização do mecanismo de persistência na aplicação Produtor-Consumidor.

com as aplicações `Processamento de` *`Tweets`*, `Detecção de Congestionamentos` e `Produtor-Consumidor`, respectivamente.

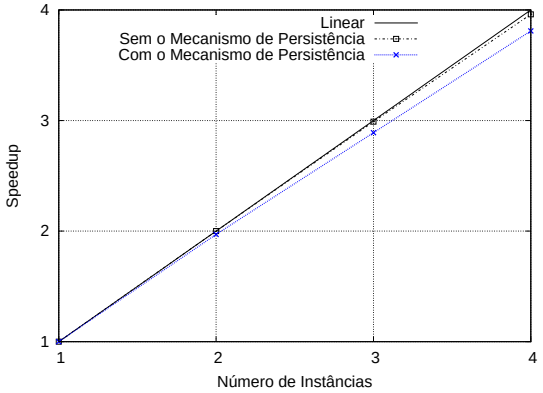
A Figura 5.6 apresenta os tempos de execução e *speedups* obtidos nos experimentos com as aplicações de teste. Pode-se observar em todas as aplicações que os resultados dos experimentos com e sem o mecanismo de persistência apresentaram comportamentos semelhantes, uma vez que o tempo de execução foi inversamente proporcional ao número de instâncias. Em ambas as versões do Watershed os *speedups* obtidos foram próximos ao *speedup* ideal (*speedup* linear). Conforme esperado, nos experimentos que utilizaram o Watershed com o mecanismo de persistência, os tempos de execução foram maiores que os obtidos utilizando a versão sem persistência. A diferença dos tempos de execução foi de no máximo 11%, 16% e 10% para as aplicações `Processamento de` *`tweets`*, `Detecção de Congestionamento` e `Produtor-Consumidor`, respectivamente.

5.3.3 Leitura de Dados Armazenados

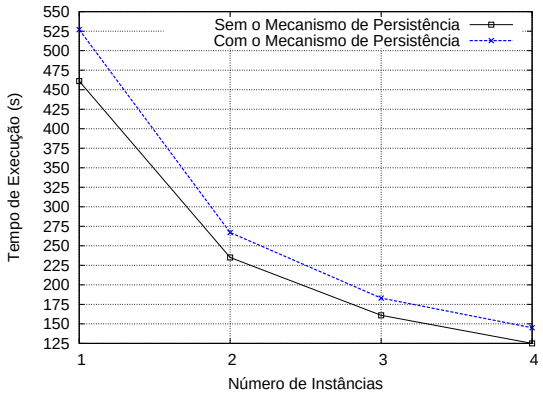
Uma série de experimentos foram realizados com a aplicação `Produtor-Consumidor` para avaliar o impacto que a leitura dos dados armazenados tem no tempo de execução das aplicações. Nos experimentos foi variada a quantidade de dados armazenados consumidos, conforme descrito a seguir:



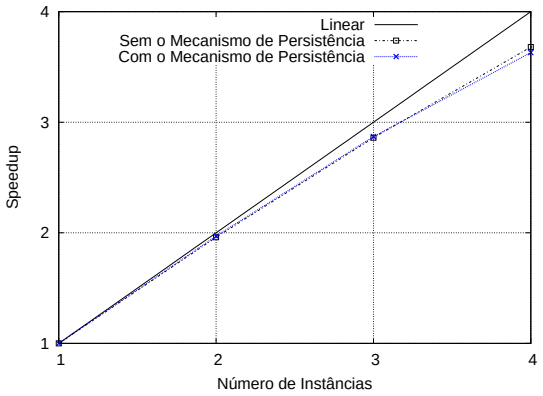
(a) Tempo de Execução da Aplicação Processamento de Tweets



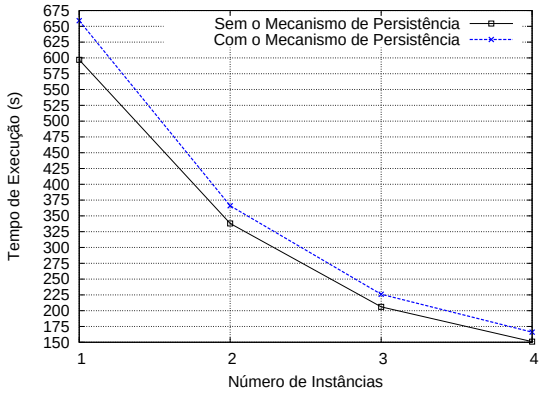
(b) Speedup da Aplicação Processamento de Tweets



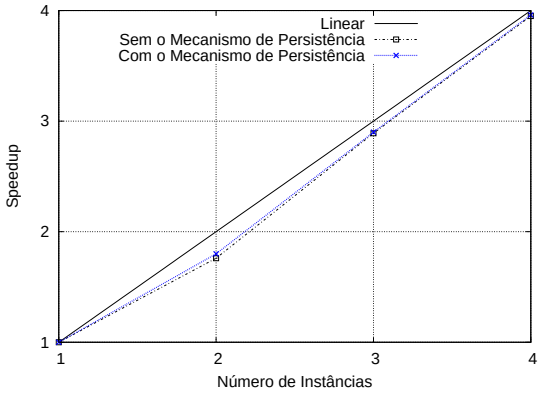
(c) Tempo de Execução da Aplicação Detecção de Congestionamentos



(d) Speedup da Aplicação Detecção de Congestionamentos



(e) Tempo de Execução da Aplicação Produtor-Consumidor



(f) Speedup da Aplicação Produtor-Consumidor

Figura 5.6. Tempos de execução e speedups das aplicações de teste.

- Configuração 1: nessa configuração o módulo `Consumidor` foi adicionado ao Watershed antes do módulo `Produtor`. Dessa maneira, todos os dados foram enviados diretamente do módulo `Produtor` para o `Consumidor`.

- Configuração 2: o módulo `Consumidor` só foi adicionado ao ambiente após o término da execução do módulo `Produtor`. Assim, todos os dados processados pelo módulo `Consumidor` foram recuperados dos bancos de dados.

Em cada experimento foram utilizadas quatro instâncias para cada módulo da aplicação e o módulo `Produtor` gerou um total de 2.000.000 dados de saída. Vale ressaltar que o tempo de processamento dos dados consumidos não é afetado pela proveniência dos dados (módulo produtor ou banco de dados). No entanto, o tempo ocioso de espera pelos dados pode ser afetado pela proveniência dos dados. Esse tempo de ociosidade tem impacto no tempo total de execução dos módulos.

Visando tornar a análise do impacto da leitura dos dados armazenados mais refinada, foram variadas as cargas de processamento dos módulos `Produtor` e `Consumidor` para criar versões rápidas e lentas. Em outras palavras, os módulos `Produtores` e `Consumidores` rápidos realizam um número bem maior de computações (i.e., cálculos de funções trigonométricas) que os respectivos módulos `Produtores` e `Consumidores` lentos. Variadas combinações dessas versões foram utilizadas nos experimentos. A Figura 5.7 apresenta os resultados obtidos utilizando as duas configurações. O tempo de processamente refere-se ao tempo total gasto apenas com o processamento dos dados (método *Process*) e o tempo de execução corresponde ao tempo desde a adição do módulo até a sua remoção.

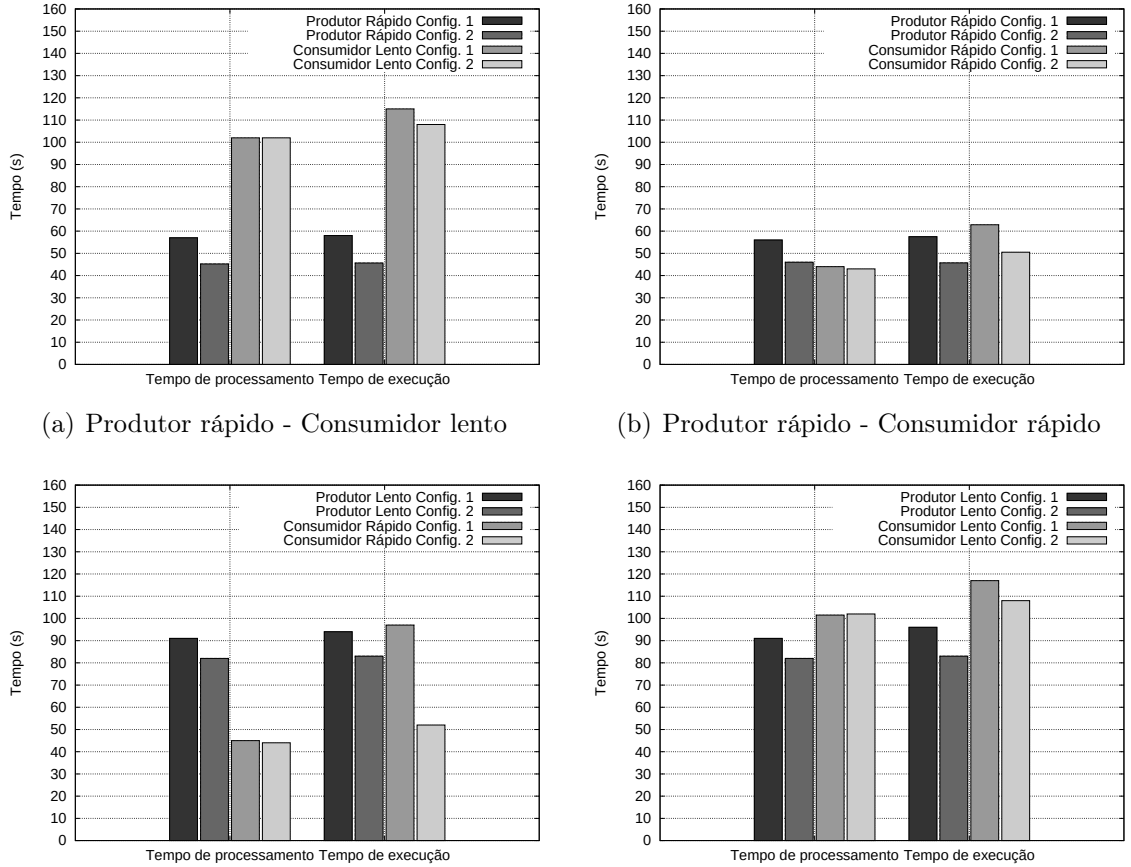
Nota-se que os tempos de processamento e execução dos módulos `Produtores`, rápidos (Figuras 5.7(a) e (b)) ou lentos (Figuras 5.7(c) e (d)), que utilizaram a Configuração 1 são sempre maiores que os tempos dos `Produtores` que utilizaram a Configuração 2. Isso ocorreu pois, na Configuração 1, os módulos têm o custo adicional de enviar os dados para as instâncias do módulo `Consumidor`. Observa-se também na subfigura 5.7(c) que o tempo de execução do módulo `Consumidor` utilizando a Configuração 1 é maior que o tempo obtido utilizando a Configuração 2. Isso ocorreu por que o tempo de ociosidade do módulo `Consumidor` é maior quando ele recebe os dados do módulo `Produtor` que é lento e, por isso, demora mais para produzir e consequentemente enviar cada dado de saída. Com esse experimento conclui-se que a leitura dos dados armazenados é feita de forma eficiente, visto que não houve impactos negativos no tempo de execução do módulo `Consumidor`.

5.3.4 Processador de Consultas

Conforme descrito anteriormente, os usuários podem definir consultas para filtrar do fluxo de entrada de um módulo apenas os dados de interesse. Para avaliar o processador de consultas utilizado no Watershed foram realizados experimentos utilizando a

aplicação `Produtor-Consumidor`. Nesses experimentos foram definidas consultas para filtrar o fluxo de entrada do módulo `Consumidor`. As consultas foram criadas para retornarem diferentes percentuais dos dados armazenados.

Em cada experimento foram utilizadas quatro instâncias de cada módulo da aplicação e o módulo `Produtor` foi configurado para gerar um total de 4.000.000 dados de saída. Para que o módulo `Consumidor` consumisse apenas dados recuperados dos bancos de dados, em todos os experimentos ele foi inserido no Watershed somente após o término da execução do módulo `Produtor`. A Figura 5.8 apresenta as médias dos tempos de execução das instâncias do módulo `Consumidor`. Como pode ser observado o processador de consultas mostrou-se eficiente, uma vez que os tempos obtidos foram proporcionais aos percentuais de dados processados.



(a) Produtor rápido - Consumidor lento

(b) Produtor rápido - Consumidor rápido

(c) Produtor lento - Consumidor rápido

(d) Produtor lento - Consumidor lento

Figura 5.7. Tempos de processamento e execução das aplicações de teste em face a variação da quantidade de dados armazenados consumidos.

5.3.5 Concorrência

Alguns experimentos foram executados para avaliar o desempenho do mecanismo de persistência frente a existência de concorrência pelo banco de dados. Para tanto foi utilizada uma versão modificada da aplicação `Produtor-Consumidor`. Inicialmente sete `Produtores`, que realizam processamentos idênticos mas possuem fluxos de saída distintos, foram executados em um mesmo computador, para que todos os dados produzidos fossem armazenadas no mesmo banco de dados. Cada módulo `Produtor` foi desenvolvido para produzir 1.000.000 dados de saída. Após o término da execução dos módulos `Produtores` foram simultaneamente adicionados ao Watershed módulos `Consumidores`, com processamento idêntico mas com tipos de fluxos de entrada díspares. Nos experimentos utilizou-se um, dois, quatro e sete módulos `Consumidores` que executaram simultaneamente em computadores distintos e com apenas uma instância cada. A Figura 5.9 apresenta os tempos obtidos. O tempo de processamente refere-se ao tempo total médio que os módulos gastaram com o processamento dos dados (método *Process*) e o tempo de execução corresponde a média das diferenças dos tempos de adição e remoção dos módulos ao Watershed.

Conforme descrito na Seção 4.4 os *daemons* de banco de dados recuperam os dados armazenados para cada módulo de forma assíncrona, entretanto ocorre concorrência pelos recursos do computador e banco de dados. Nota-se que o tempo de

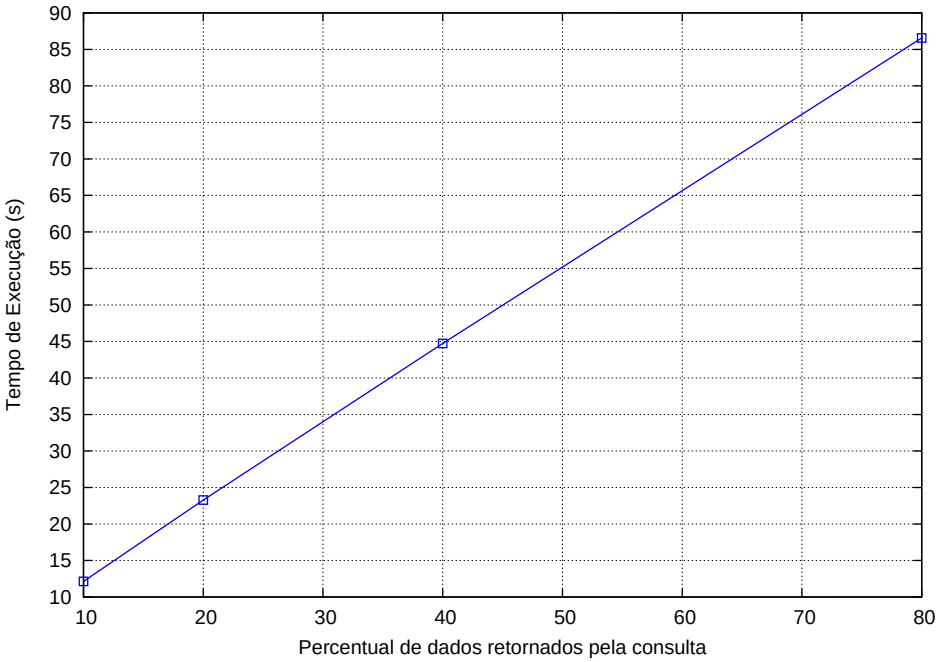


Figura 5.8. Tempos de execução médios do módulo Consumidor em face a utilização de filtros para seu fluxo de entrada.

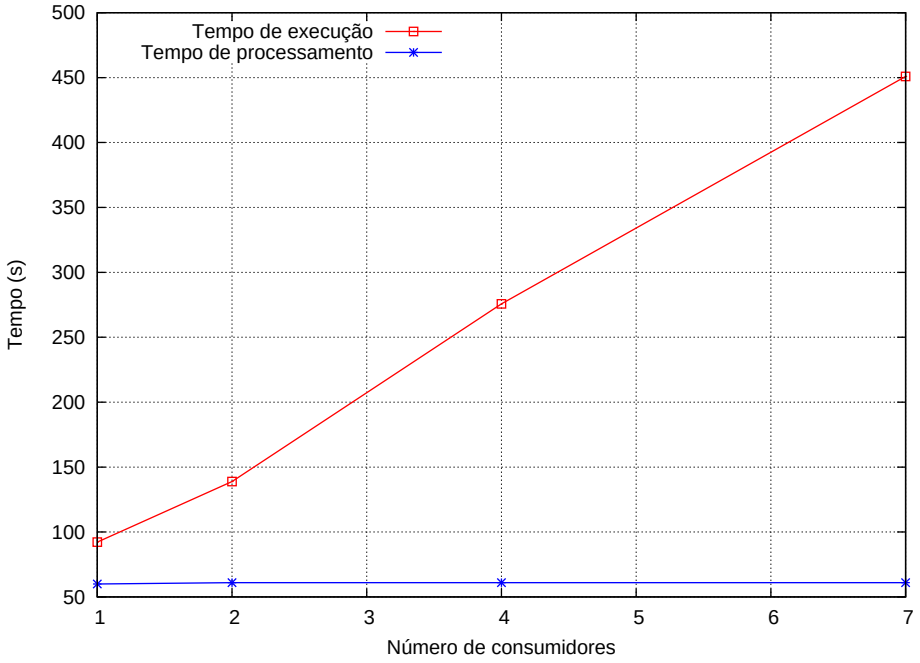


Figura 5.9. Tempos de execução médios dos módulos Consumidores em face a concorrência pelo banco de dados.

execução médio dos módulos aumenta sublinearmente em relação ao número de instâncias de módulos que simultaneamente consultam dados a um *daemon* de banco de dados. No entanto esse é o pior caso de influência deste efeito no tempo de execução dos módulos, pois todos dependem exclusivamente de um único *daemon* para progredir.

Capítulo 6

Conclusões e Trabalhos Futuros

Em conjunto com outros pesquisadores do laboratório e-SPEED foi construído um ambiente de execução de alto desempenho, denominado Watershed, para o desenvolvimento e execução de aplicações de processamento de fluxos de dados. O Watershed explora os recursos computacionais disponíveis, com o objetivo de alcançar maior eficiência das aplicações que executam sobre ele, abstraindo as dificuldades inerentes ao desenvolvimento de aplicações distribuídas e paralelas. Assim, os usuários podem se concentrar apenas no desenvolvimento dos módulos de processamento de suas aplicações, ficando a cargo do ambiente criar as instâncias dos módulos, conectá-los dinamicamente de acordo com uma abordagem dirigida a dados, gerenciar o roteamento das mensagens, fazer o controle de fluxo, enfim, coordenar toda a execução que ocorre de forma paralela e distribuída. O Watershed é um ambiente de código aberto cujas principais características são: (i) permitir o desenvolvimento e a execução de aplicações com topologia dinâmica; e (ii) possibilitar a execução simultânea de múltiplas aplicações que podem compartilhar resultados intermediários. Essas características diferem o Watershed da maioria dos sistemas de processamento de fluxos de dados descritos na literatura.

A principal contribuição deste trabalho foi a construção do mecanismo de persistência de fluxos de dados para o Watershed. O mecanismo implementado possibilita que o Watershed seja mais genérico e flexível, pois ele armazena os dados produzidos pelos módulos de forma transparente e possibilita que dados históricos e atuais possam ser consumidos pelos módulos de processamento. Além disso, o mecanismo é distribuído, manipula dados semiestruturados e permite que os módulos filtrem dos seus fluxos de entrada apenas os dados atuais ou históricas de seu interesse. Uma vez que o Watershed possibilita que diferentes aplicações possam compartilhar resultados intermediários, o mecanismo disponibiliza métodos para que os usuários possam ter

conhecimento de todos os fluxos de dados disponíveis. O formato de representação de dados utilizado pelo mecanismo de persistência e adotado para o Watershed facilita o desenvolvimento dos módulos de processamento, visto que diversos métodos estão disponíveis para tornar a manipulação dos dados mais simples. Dessa maneira todos os requisitos levantados na fase de projeto foram completamente atendidos.

Os resultados obtidos na avaliação experimental mostram que o Watershed é escalável com o mecanismo de persistência. Observa-se que o custo adicionado ao tempo de execução das aplicações para persistir os dados é baixo. Para as aplicações e configurações avaliadas o custo médio foi de 13%. Os resultados também mostram que o processador de consultas do mecanismo de persistência é eficiente no processamento de consultas com quantidades variadas de dados servidos.

6.1 Trabalhos Futuros

Algumas oportunidades de trabalhos futuros que podem ser desenvolvidos para tornar o mecanismo de persistência e o Watershed mais completos são apresentados a seguir:

- Desenvolver uma estratégia para tratar o problema da remoção automática dos dados armazenados, pois uma vez que enormes quantidades de dados são continuamente armazenados, seria interessante desenvolver uma estratégia para tratar esse problema.

- Projetar um mecanismo de proveniência de dados com base no mecanismo de persistência desenvolvido que permita o rastreamento das transformações realizadas nos dados.

- Permitir ao usuário decidir quais fluxos de dados devem ser persistidos para dessa forma aliviar o mecanismo de persistência e os bancos de dados.

- Incluir um mecanismo de segurança que controle as permissões de acesso de fluxos de dados persistidos para proteção e compartilhamento controlado dos dados.

- Explorar técnicas de *sharding* e de replicação para melhorar o desempenho da leitura dos fluxos de dados em bancos de dados sobrecarregados.

- Avaliar o Watershed com o mecanismo de persistência em cenários com diferentes ocupações do banco de dados. Realizar também avaliações de latência e banda.

- Projetar um mecanismo de tolerância a faltas para o Watershed.

Referências Bibliográficas

Acharya, A.; Uysal, M. & Saltz, J. (1998). Active disks: programming model, algorithms and evaluation. Em *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 81–91.

Agha, G. (1986). *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA.

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. Em *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms*, pp. 27–37.

Andrade, H.; Kurc, T.; Sussman, A. & Saltz, J. (2001). Efficient execution of multiple query workloads in data analysis applications. Em *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pp. 53–53.

Banker, K. (2011). *MongoDB in Action*. Manning Publications, Greenwich, CT, USA.

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 Comput.*, 28(5):827–859.

Beynon, M.; Ferreira, R.; Kurc, T.; Sussman, A.; Saltz, J. & Medical, J. H. (2000). DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems. Em *Proceedings 8th Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, pp. 119–133, College Park, MD.

Beynon, M. D.; Kurc, T.; Catalyurek, U.; Chang, C.; Sussman, A. & Saltz, J. (2001). Distributed processing of very large datasets with datacutter. *Parallel Comput.*, 27(11):1457–1478.

Blount, M.; Ebling, M.; Eklund, J.; James, A.; McGregor, C.; Percival, N.; Smith, K. & Sow, D. (2010). Real-time analysis for intensive care: Development and deployment of the artemis analytic system. *Engineering in Medicine and Biology Magazine*, 29(2):110 –118.

Botan, I.; Alonso, G.; Fischer, P. M.; Kossmann, D. & Tatbul, N. (2009). Flexible and scalable storage management for data-intensive stream processing. Em *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pp. 934–945.

BSON (2012). Binary JSON. `http://bsonspec.org/`.

Carney, D.; Çetintemel, U.; Cherniack, M.; Convey, C.; Lee, S.; Seidman, G.; Stonebraker, M.; Tatbul, N. & Zdonik, S. (2002). Monitoring streams: a new class of data management applications. Em *Proceedings of the 28th International Conference on Very Large Data Bases*, pp. 215–226.

Chakravarthy, S. & Jiang, Q. (2009). *Stream Data Processing: A Quality of Service Perspective - Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Springer.

Chodorow, K. & Dirolf, M. (2010). *MongoDB: The Definitive Guide*. O'Reilly Media.

Cugola, G. & Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15.

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. Em *Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, pp. 159–167, Rio de Janeiro, Brazil. IEEE Computer Society.

Fireman, D.; Teodoro, G.; Cardoso, A. & Ferreira, R. (2008). A reconfigurable run-time system for filter-stream applications. Em *Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing*, pp. 149–156, Campo Grande, Mato Grosso do Sul, Brazil. IEEE Computer Society.

Flynn, I. (2002). *Introdução aos Sistemas Operacionais*. Thomson Learning (Portug).

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. Em *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. Em *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pp. 1123–1134, New York, NY, USA. ACM.

Gomide, J.; Veloso, A.; Meira Jr., W.; Almeida, V.; Benevenuto, F.; Ferraz, F. & Teixeira, M. (2011). Dengue surveillance based on a computational model of spatio-temporal locality of twitter. Em *Proceedings of the ACM WebSci*.

Hildrum, K.; Douglis, F.; Wolf, J. L.; Yu, P. S.; Fleischer, L. & Katta, A. (2008). Storage optimization for large-scale distributed stream-processing systems. *Trans. Storage*, 3(4):1–28.

Hilley, D. & Ramachandran, U. (2009). Persistent temporal streams. Em *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, pp. 1–20, New York, NY, USA.

Kargupta, H. & Han, J. (2009). *Next Generation of Data Mining*. CRC Press.

Lin, J.; Snow, R. & Morgan, W. (2011). Smoothing techniques for adaptive online language models: topic tracking in tweet streams. Em *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 422–429.

Liu, J. & Panda, D. K. (2004). Implementing efficient and scalable flow control schemes in mpi over infiniband. Em *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, USA.

Migliavacca, M.; Papagiannis, I.; Eyers, D. M.; Shand, B.; Bacon, J. & Pietzuch, P. (2010). Defcon: high-performance event processing with information security. Em *Proceedings of the USENIX annual technical conference*, pp. 1–15, Berkeley, CA, USA.

Moreau, L. (2010). The foundations for provenance on the web. *Found. Trends Web Sci.*, 2:99–241.

Neumeyer, L.; Robbins, B.; Nair, A. & Kesari, A. (2010). S4: Distributed stream computing platform. Em *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, pp. 170–177, Sydney, Australia. IEEE Computer Society.

Ramos, T. L. A. S.; Oliveira, R. S.; Carvalho, A. P. d.; Ferreira, R. A. C. & Meira Jr., W. (2011). Watershed: A high performance distributed stream processing system. Em *Proceedings of the 23rd International Symposium on Computer Architecture and High Performance Computing*, pp. 191–198, Vitoria, ES, Brazil.

Santos, W.; Pappa, G.; Meira Jr., M.; Guedes, D.; Veloso, A.; Almeida, V.; Pereira, A.; Guerra, P.; Silva, A.; Mourão, F.; Magalhães, T.; Machado, F.; Cherchiglia, L.; Simões, L.; Batista, R.; Arcanjo, F.; Brunoro, G.; Mariano, N.; Magno, G.; Ribeiro, M. T. C.; Teixeira, L.; Silva, A. S. d.; Reis, B. W. & Silva, R. H. (2010). Observatório da web: Uma plataforma de monitoração, síntese e visualização de eventos massivos em tempo real. Em *Anais do XXXVII Seminário Integrado de Hardware e Software*, pp. 110–120, Belo Horizonte, MG, Brazil.

Teodoro, G.; Fireman, D.; Guedes Neto, D. O.; Meira Jr., W. & Ferreira, R. (2008). Achieving multi-level parallelism in the filter-labeled stream programming model. Em *Proceedings of the 37th International Conference on Parallel Processing*, pp. 287–294.

Teodoro, G.; Hartley, T. D. R.; Çatalyürek, Ü. V. & Ferreira, R. (2010). Run-time optimizations for replicated dataflows on heterogeneous environments. Em *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 13–24.

Apêndice A

Aplicação Exemplo

Para ilustrar o uso do Watershed, este apêndice apresenta a descrição detalhada do desenvolvimento de uma aplicação nesse ambiente.

A.1 Descrição da Aplicação

A função da aplicação exemplo é criar e manter uma nuvem de *tags* com as palavras que aparecem em *tweets* relacionados a Dilma. Ela pode ser decomposta em quatro módulos de processamento:

- *Collector*: módulo responsável por fazer a coleta de *tweets*. Seu fluxo de saída é denominado *Tweets*.
- *StopwordsRemover*: esse módulo consume dados do fluxo *Tweets*. Desse fluxo são filtrados apenas os *tweets* relacionados a Dilma. A função do módulo *StopwordsRemover* é remover todas as *stopwords* presentes no texto de um tweet. O fluxo de saída produzido é chamado *Words*.
- *WordCounter*: esse módulo é responsável por contar o número de ocorências das palavras que são recebidas do fluxo *Words*. O fluxo produzido é denominado *FrequencyWord*.
- *TagCloudGenerator*: módulo responsável pela criação e manutenção de uma nuvem de *tags*. Esse módulo recebe as palavras e suas frequências do fluxo *FrequencyWord*.

As aplicações construídas sobre o Watershed são compostas de duas partes: (i) descrição da topologia e (ii) descrição do processamento.

A.2 Descrição da Topologia

Para cada módulo de processamento é associado um arquivo descritor de topologia escrito no formato XML. As Figuras A.1 e A.2 mostram os arquivos descritores de topologia dos módulos da aplicação exemplo. A estrutura do fluxo de saída de cada módulo de processamento é descrita por meio de documentos JSON, conforme apresentado na Figura A.3.

A topologia de uma aplicação Watershed é dinamicamente determinada pelos fluxos que cada módulo consome e produz. A Figura A.4 apresenta a topologia da aplicação exemplo.

A política de recebimento de mensagens definida para o módulo *WordCounter* é a fluxo rotulado. Para cada módulo que utiliza essa política deve ser desenvolvida uma função *hash* para determinar a instância de destino de cada mensagem enviada para esse módulo. A Figura A.5 apresenta a função *hash* do módulo *WordCounter*. Para o módulo *TagCloudGenerator* foi definida a política *broadcast*. Entretanto, vale ressaltar que quando um módulo possui apenas uma instância, como é o caso desse módulo, todas as políticas possuem o mesmo comportamento, sendo portanto equivalentes.

A.3 Descrição do Processamento

Ao integrar um novo módulo de processamento no Watershed, o programador deve implementar três métodos. O primeiro é o construtor do módulo, que inicializa o estado

```
<processing_module>
 <global
  name = "Collector"
  library = "/w/collector.so"
  instances = "2">
 </global>
 <output name = "Tweets"
  structure = "/w/collector_s"/>
 <demands>
  <demand name = "Twitter_API"/>
 </demands>
</processing_module>
```

```
<processing_module>
 <global
  name = "TagCloudGenerator"
  library = "/w/tag_cloud_generator.so"
  instances = "1">
 </global>
 <inputs>
  <input
   name = "FrequencyWord"
   query = "none"
   reader_persisted_data = "true"
   initial_date = "none"
   policy = "broadcast"/>
 </inputs>
 <demands>
   <demand name = "common"/>
 </demands>
</processing_module
```

Figura A.1. Arquivos XML de descrição da topologia dos módulos de processamento. (Esquerda) Collector. (Direita) TagCloudGenerator.

```
<processing_module>
 <global
  name = "WordCounter"
  library = "/wt/word_counter.so"
  instances = "5">
 </global>
 <inputs>
  <input
   name = "Words"
   query = "none"
   reader_persisted_data = "true"
   initial_date = "none"
   policy = "labeled"
   policy_function_file="/w/word_counter_l.so"/>
 </inputs>
 <output name = "FrequencyWord"
  structure = "/w/word_counter_s"/>
 <demands>
  <demand name = "common"/>
 </demands>
</processing_module>
```

```
<processing_module>
 <global
  name = "StopwordRemover"
  library = "/wt/stopword_remover.so"
  instances = "5"
  arguments = "-i /w/stopWords.txt">
 </global>
 <inputs>
  <input
   name = "Tweets"
   query="{text:{$regex:‘.*dilma’,$options:‘i’}}"
   reader_persisted_data = "true"
   initial_date = "none"
   policy = "round_robin"/>
 </inputs>
 <output name = "Words"
  structure = "/w/stopword_remover_s"/>
 <demands>
  <demand name = "common"/>
 </demands>
</processing_module>
```

Figura A.2. Arquivos XML de descrição da topologia dos módulos de processamento. (Esquerda) WordCounter. (Direita) StopwordRemover.

```
{"Tweets": {
    "text":{
        "type": "string",
        "required": "true"
     }
}}
```

```
{"Words": {
    "words":{
        "type": "string",
        "required": "true"
     }
}}
```

```
{"FrequencyWord": {
    "words":{
        "type": "string",
        "required": "true"
    },
    "frequency":{
        "type": "int",
        "required": "true"
     }
}}
```

Figura A.3. Estrutura dos fluxos de saída produzidos pelos módulos da aplicação exemplo. (Esquerda) Módulo Reader. (Meio) Módulo StopwordRemover. (Direita) Módulo WordCounter.

interno do módulo antes dele iniciar qualquer processamento. O segundo método, chamado *process*, determina as ações que um módulo deve realizar sempre que uma mensagem for recebida. O último método é o destrutor do módulo, que é chamado quando a terminação do módulo é solicitada. Essa solicitação de terminação pode ser feita de duas formas: i) quando o módulo explicitamente indica a terminação, através da chamada do método *TerminateModule*; ii) quando o programador solicita, via console, a remoção do módulo. As Figuras A.6 e A.7 mostram as implementações desses métodos para os módulos *StopwordsRemover* e *WordCounter*. Como o código dos demais módulos da aplicação exemplo são mais extensos e complexos eles não serão apresentados, entretanto as implementações apresentadas são suficientes para ilustrar o uso do Watershed.

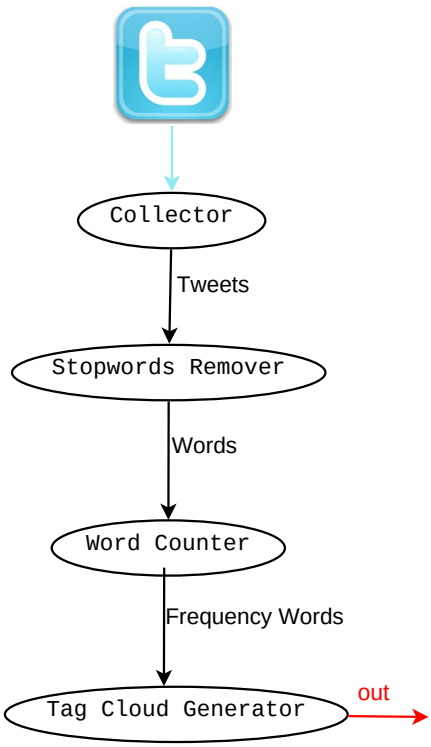


Figura A.4. Topologia da aplicação.

```
int WordCounterLS::GetLabel ( Message& message, int total_instances ){
  Json* doc = message.GetJson();
  string word = doc.GetStringField("word");
  int sum = 0;
  for ( uint i = 0; i < word.length (); ++i ) {
    sum += word.at ( i );
  }
  return abs ( sum % total_instances );
}
```

Figura A.5. Implementação da função hash utilizada na política de recebimento de mensagens do módulo WordCounter

```
StopwordRemover::StopwordRemover() {
  input_file_name_ = GetArgument( "i" );
  LoadStopWords();
}

StopwordRemover::~StopwordRemover() {
  stopwords_.clear();
}

bool StopwordRemover::IsStopWord( string word ) {
  if ( stopwords_.find(word) != stopwords_.end() ){
    return true;
  }
  return false;
}

void StopwordRemover::LoadStopWords(){
   //Lê arquivo com as stopwords
   ...
}

void StopwordRemover::Process( Message& message ){
  Json* doc = message.GetJson();
  string text = doc.GetStringField("text");
  delete(doc);
  vector <string> tweet_words = Util::TokenizeString(" ", text);
  Message output_message;
  for (int i = 0; i < (int) tweet_words.size(); ++i){
    if ( !IsStopWord(tweet_words[i]) ) {
      Json doc_out;
      doc_out.AppendStringField( "word", tweet_words[i] );
      output_message.SetJson( doc_out );
      Send( output_message );
    }
  }
}
```

Figura A.6. Implementação do módulo StopwordRemover

```
WordCounter::~WordCounter(){
  count_map_.clear ();
}

void WordCounter::Process( Message& message ){
  Json* doc = message.GetJson();
  string word = doc.GetStringField("word");
  delete(doc);
  if(count_map_.find(word) == count_map_.end()){
    count_map_[word] = 1;
  }
  else {
    ++count_map_[word];
  }
  if(count_map_[word] % 25 == 0){
    Message output_message;
    Json doc_out;
    doc_out.AppendStringField("word", word);
    doc_out.AppendIntField("frequency", count_map_[word]);
    output_message.SetJson(doc_out);
    Send( output_message );
  }
}
```

Figura A.7. Implementação do módulo WordCounter