WATERSHED-NG: UM SISTEMA DISTRIBUÍDO E EXTENSÍVEL PARA O PROCESSAMENTO DE FLUXOS DE DADOS

RODRIGO CAETANO DE OLIVEIRA ROCHA

WATERSHED-NG: UM SISTEMA DISTRIBUÍDO E EXTENSÍVEL PARA O 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: DORGIVAL GUEDES

Belo Horizonte

Julho de 2015

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Rocha, Rodrigo Caetano de Oliveira.

R672w Watershed-ng: um sistema distribuído e extensível para o processamento de fluxos de dados / Rodrigo Caetano de Oliveira Rocha.— Belo Horizonte, 2015.
xvi, 48 f. : il. ; 29cm.

Dissertação (Mestrado) - Universidade Federal de Minas Gerais Departamento de Ciência da Computação.

Orientador: Dorgival Olavo Guedes Neto:

1. Computação - Teses. 2. Programação paralela (Computação) - Teses. 3. Sistemas distribuídos - Teses 4. Big data. I. Orientador II. Título.

519.6*31(043)

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

FOLHA DE APROVAÇÃO

Watershed-ng: um sistema distribuído e extensível para o processamento de fluxos de dados

RODRIGO CAETANO DE OLIVEIRA ROCHA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

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

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

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

Belo Horizonte, 17 de julho de 2015.

Resumo

Plataformas de processamento de dados massivos (*a.k.a. big-data*) permitem aos usuários expressarem as computações utilizando abstrações, tal como o MapReduce, simplificando a extração de paralelismo das aplicações. A maioria das plataformas, no entanto, não permitem que os usuários especifiquem como a comunicação entre os nós de computação deve acontecer: esse elemento geralmente está implementado diretamente no sistema de tempo de execução (RTS), com alto grau de acoplamento, dificultando a realização de mudanças. Neste trabalho descrevemos a plataforma Watershed-ng, uma re-engenharia do Watershed, uma plataforma baseada no modelo filtro-fluxo e originalmente focada no processamento de fluxos contínuos de dados. Similar à outros ambientes de dados massivos, o sistema do Watershed oferecia uma abstração orientada à objetos para descrever apenas o padrão de processamento das aplicações. Isolando a funcionalidade dos canais de fluxos de dados em classes de primeira ordem, se tornou possível desenvolver, combinar e reutilizar diversos padrões de comunicação e componentes para o tratamento do fluxo de dados. Aplicações desenvolvidas com o Watershed-ng, integrado ao ambiente Hadoop, apresentaram melhorias significativas de desempenho, bem como uma redução em tamanho de código de até 50%, quando comparado com as plataformas predecessoras.

Abstract

Most high-performance data processing (*a.k.a.* big-data) systems allow users to express their computation using abstractions (like MapReduce) that simplify the extraction of parallelism from applications. Most frameworks, however, do not allow users to specify how communication must take place: that element is deeply embedded into the run-time system (RTS) abstractions, making changes hard to implement. In this work we describe Wathershed-ng, our re-engineering of the Watershed system, a framework based on the filter-stream paradigm and originally focused on continuous stream processing. Like other big-data environments, Watershed provided object-oriented abstractions to express computation (filters), but the implementation of streams was an RTS element. By isolating stream functionality into appropriate classes, combination of communication patterns and reuse of common message handling functions (like compression and blocking) become possible. The new architecture even allows the design of new communication patterns, for example, allowing users to choose MPI, TCP or shared memory implementations of communication channels as their problem demands. Applications designed for the new interface showed reductions in code size on the order of 50% and above in some cases. The performance results also showed significant improvements, since some implementation bottlenecks were removed in the re-engineering process.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

A explosão de dados disponíveis para os pesquisadores nos dias atuais levou ao crescimento da área de processamento de dados de alto desempenho, também conhecida como *big-data*. Para obter os melhores resultados nesta área, as ferramentas têm de estar disponíveis para os especialistas no domínio dos dados. No entanto, tais especialistas raramente possuem também experiência em programação paralela, criando a necessidade de desenvolver plataformas que possam explorar o paralelismo intrínseco à aplicação com uma API simples. Modelos com abstrações simplificadas que favorecem a generalização e produtividade em relação à eficiência da computação têm sido amplamente adotados em ambos os segmentos de pesquisa e da indústria [Hall et al., 2013]. O modelo de programação MapReduce [Dean & Ghemawat, 2008], originalmente desenvolvido pela Google, com sua implementação *open-source*, Hadoop [White, 2009], são, certamente, um dos exemplos mais conhecidos de tais plataformas. Com a popularidade do Hadoop, a vasta utilização desse ambiente expôs algumas limitações do ambiente, criando a necessidade de um processo de modularização, pelo projeto do YARN, desacoplando o modelo de programação da infra-estrutura de gerenciamento de recursos, além de permitir a integração com outras aplicações [Vavilapalli et al., 2013].

A maioria dessas plataformas, no entanto, não permitem que os usuários especifiquem como a comunicação entre os nós de computação deve acontecer: esse elemento geralmente está implementado diretamente no sistema de tempo de execução (RTS, ou *Run Time System*), com alto grau de acoplamento, dificultando a realização de mudanças.

Watershed [de Souza Ramos et al., 2011] também é um exemplo de plataforma desenvolvida com objetivos similares. Watershed é um sistema distribuído para o processamento de grandes volumes de fluxos de dados, derivado do Anthill [Ferreira et al., 2005], uma plataforma desenvolvida anteriormente para o processamento em

batch. Ambas as plataformas foram inspiradas no modelo de fluxo de dados (*data-flow*). Sistemas de processamento de fluxo de dados compreendem um conjunto de componentes que computam em paralelo (chamados de filtros) e que se comunicam através de fluxos de dados, ou seja, os filtros obtêm os dados através dos canais de comunicação, realizam o processamento dos dados recebidos e enviam pelos canais de saída os dados produzidos durante o processamento do filtro. Em sua forma original, a plataforma Watershed apresentava os canais de comunicação como caixas-pretas, isto é, elementos que não podem ser alterados pelo usuário. Além disso, sua orientação para o processamento de fluxos contínuos de dados tornou difícil sua utilização para as aplicações mais tradicionais de processamento em *batch* orientada a arquivos, algo que seu antecessor, o Anthill, permitia.

Para mudar isso, realizamos a re-engenharia da plataforma Watershed, tornando a abstração dos canais de fluxos de dados extensíveis pelo programador. Ao isolar a funcionalidade dos fluxo em classes apropriadas, se torna possível combinar padrões de comunicação e reutilizar elementos comuns de tratamento de mensagens, como componentes de compressão e agrupamento. Por exemplo, os usuários devem ser capazes de adicionar transformações para um fluxo de dados de uma forma elegante, sem a necessidade de desenvolver novos filtros de processamento ou alterar o código da aplicação existente. Um caso comum é o de agregação de mensagens: em muitas aplicações, para reduzir a sobrecarga da rede devido à transmissão de um grande volume de mensagens pequenas, muitas vezes é possível obter ganho de desempenho agrupando um grande número de itens de dados em uma única mensagem a ser enviada através da rede. Na versão original do Watershed, cada desenvolvedor teria que acrescentar o código de agregação (e sua contraparte de desagregação) internamente ao código da aplicação. Na versão re-projetada, Watershed-ng, é possível reutilizar um elemento comum de agregação/desagregação que pode ser adicionado diretamente ao fluxo, desacoplado do código do aplicação.

A nova arquitetura da plataforma deve permitir igualmente a criação de novos padrões de comunicação, por exemplo, permitir que os usuários escolham entre MPI, TCP ou implementações de memória compartilhada de canais de comunicação de acordo com a demanda de seu problema. Isso ajudaria também a tornar o Watershed mais simples de usar no desenvolvimento de aplicações em *batch* ou orientada a arquivos, pois fluxos poderiam ser ampliados para atender melhor a essa forma de processamento.

1.1 Objetivos

Nosso objetivo com este projeto é que os usuários fossem capazes de estender a plataforma com outros elementos de comunicação, melhores adaptadas aos seus algoritmos. Para isso, realizaremos a re-engenharia da plataforma Watershed, desenvolvendo uma abstração modular para os fluxos de dados, oferecendo uma interface extensível aos componentes de comunicação e processamento.

No processo de realizar a re-engenharia da plataforma Watershed, pretendemos também integrá-la ao ecossistema Hadoop, de maneira a permitir sua coexistência com outras plataformas de processamento em um mesmo ambiente de execução. Pretende-se também oferecer uma biblioteca padrão que ofereça alguns fluxos de dados que implementem padrões de comunicação que sejam comuns à diversas aplicações, incluindo àqueles existentes nas implementações anteriores do Watershed e Anthill.

1.2 Contribuições

Como principais contribuições deste trabalho, destacamos:

- re-engenharia, integrando as qualidades do Anthill e Watershed;
- modularização da abstração de comunicação com componentes de reutilização para ações usuais;
- aplicação dessa modularização na integração do sistema com a plataforma Hadoop, em particular, a integração com o escalonador YARN e o sistema de arquivos HDFS.

1.3 Organização

O restante deste trabalho descreve os conceitos básicos relacionados e trabalhados anteriores relevantes, as nossas decisões de projeto durante o processo de re-engenharia, os principais aspectos da nova arquitetura RTS e alguns resultados de desempenho. Esses resultados mostram que aplicações desenvolvidas utilizando a nova API apresentaram reduções no tamanho do código da ordem de 50%, em alguns casos até acima. Além disso, o processo de re-engenharia nos permitiu remover alguns gargalos inerentes à implementação, resultando em melhorias significativas de desempenho.

A dissertação está organizada como segue. O capítulo 2 apresenta os principais conceitos relacionados necessários para o desenvolvimento deste trabalho, bem como

os principais trabalhos de pesquisa relacionados. O capítulo 3 apresenta a abstração do Watershed-ng, descrevendo o funcionamento dos filtros, fluxos de dados e a conexão entre eles. Esse mesmo capítulo apresenta também uma aplicação de exemplo implementada na plataforma Watershed-ng. A arquitetura do Watershed-ng, incluindo sua integração com o ecossistema Hadoop, é descrita no capítulo 4. Por fim, os capítulos 5 e 6 apresentam os resultados experimentais e conclusões, respectivamente.

Capítulo 2

Conceitos relacionados

Para melhor entender este trabalho devemos primeiramente entender a estrutura e relação entre os componentes do ecossistema Hadoop importantes para a implementação do Watershed-ng, bem como os trabalhos de pesquisa relacionados.

2.1 A plataforma Hadoop

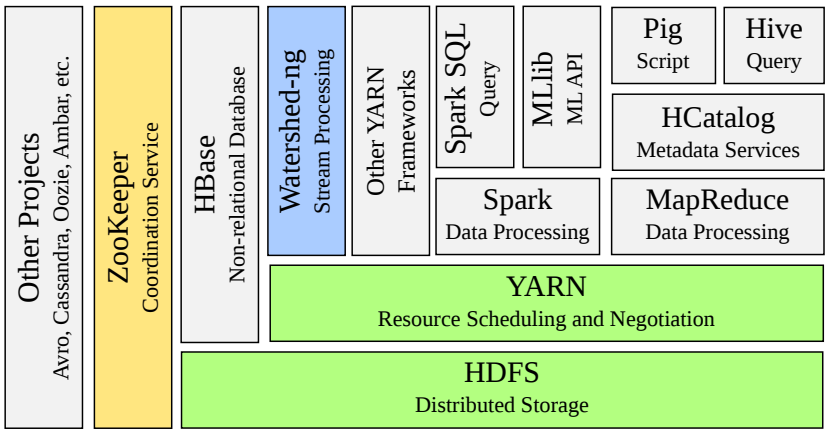

Figura 2.1. Ilustração dos componentes que integram o ecossistema Hadoop, incluindo o papel do Watershed-ng em relação à todo o ecossistema.

Essa seção apresenta os fundamentos básicos dos principais componentes utilizados no desenvolvimento da nova plataforma do Watershed-ng. Para essa nova implementação do Watershed-ng, o sistema foi integrado com o ecossistema Hadoop, visando possibilitar maior integração com aplicações já existentes que utilizam outros sistemas integrados ao Hadoop bem como permitir também a coexistência de aplicações em execução em um mesmo ambiente de *cluster*. A integração com o ecossistema Ha-

doop se dá principalmente ao integrar o Watershed ao sistema de arquivos distribuído, HDFS (Hadoop Distributed File System), e ao serviço de gerenciamento de recursos e processos do *cluster*, YARN. Outro serviço utilizado por diversos componentes do ecossistema Hadoop e também pelo Watershed-ng é o serviço de coordenação de processos e aplicações distribuídas, chamado ZooKeeper. A figura 2.1 apresenta os principais componentes do ecossistema Hadoop, incluindo serviços auxiliares e também o papel do Watershed-ng em relação à todo o ecossistema.

2.1.1 Serviço de gerenciamento de recursos e processos distribuídos

Idealmente, aplicações distribuídas decidem dinamicamente em qual nó de computação cada tarefa será executada, monitorando o estado de cada tarefa durante a execução. Monitorar o estado das tarefas possibilita, por exemplo, saber se as tarefas foram concluídas, bem tomar medidas de tolerância à falhas sempre que necessário. Para ser capaz de realizar um escalonamento inteligente dessas tarefas distribuídas, considerando maior nível de detalhes do ambiente de execução, é necessário também gerenciar os recursos disponíveis oferecidos pelos nós de computação.

O YARN, escalonador de recursos do ecossistema Hadoop, oferece o serviço de uma plataforma básica para a gestão das aplicações distribuídas em um mais alto nível de abstração. Permitindo assim que diferentes aplicações distribuídas possam co-existir em um mesmo ambiente de execução em *cluster*.

A principal ideia do YARN é particionar suas duas principais responsabilidades — gerenciamento de recursos e escalonamento de tarefas – em componentes separados: um gerenciador global de recursos (*ResourceManager* - RM) e uma tarefa mestre de gerenciamento por aplicação (*ApplicationMaster* - AM). O sistema YARN é composto pelo *ResourceManager* e uma tarefa escrava em cada nó do cluster, chamada de *NodeManager* (NM) [Murthy et al., 2013].

O *ApplicationMaster*, que é o módulo desenvolvido especificamente por cada aplicação, é o processo que coordenada a execução da própria aplicação no ambiente do *cluster*.

O *NodeManager* é responsável por gerenciar a execução dos *containers* locais, monitorando suas utilizações de recursos — tais como CPU, memória, disco e rede — enviando para o RM informações sobre os *containers*.

O *ResourceManager* é a autoridade máxima que arbitra a divisão dos recursos entre todos os aplicativos em execução no ambiente. O *NodeManager* tem um componente para o escalonamento de recursos, que é responsável pela alocação de recursos

para os vários aplicativos em execução sujeito às restrições principais de capacidade, prioridade, e outros fatores. O escalonador não realiza nenhum monitoramento ou rastreamento para a aplicação, sem garantias de reiniciar tarefas que não são adequadamente finalizadas devido a qualquer falha da própria aplicação ou falhas de hardware. O escalonador executa seu escalonamento baseado em recursos requisitados pelo aplicativo usando a noção abstrata de um *container* de recursos, que incorpora dimensões de recursos, tais como memória, CPU, disco e rede.

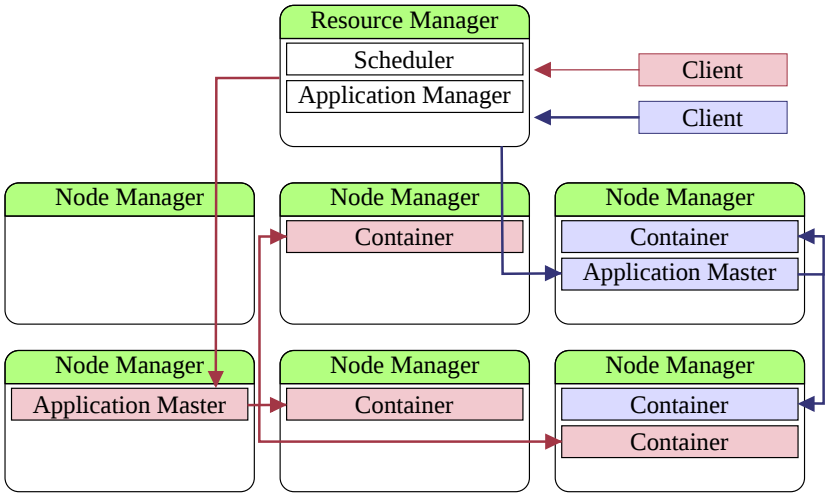

Figura 2.2. Exemplo ilustrando o funcionamento da arquitetura do YARN gerenciando duas aplicações distintas em um ambiente de cluster.

O Watershed-ng se integra ao YARN, tirando proveito dos serviços de gerenciamento de recursos e processos distribuídos, que são utilizados principalmente para escalonar, executar e monitorar cada componente de processamento de uma aplicação em Watershed, isto é, as instâncias dos filtros bem como os componentes que integram os fluxos de dados.

2.1.2 Sistema de arquivos distribuído

Sistemas para processamento de dados massivos lidam com coleções de dados que podem esgotar toda a capacidade de armazenamento de uma única máquina. Além disso, processar tais coleções de dados demanda grande poder computacional. Por esse motivo, grandes coleções de dados são armazenadas de maneira distribuídas em diversos nós computacionais, permitindo que grandes volumes de dados sejam armazenados, além da capacidade individual de cada máquina, bem como o processamento e acesso distribuído de tais coleções de dados.

O *Hadoop Distributed File System* (HDFS) [Shvachko et al., 2010] foi baseado principalmente no *Google File System* (GFS) [Ghemawat et al., 2003], que é um sistema distribuído de arquivos que proporciona tolerância à falhas mesmo enquanto executando sobre hardware considerados como produtos de conveniência. Nesse sistema, arquivos são escritos sequencialmente quando criados, ou dados são acrescentados sequencialmente ao final de um arquivo já existente, pela operação de *append*[1].

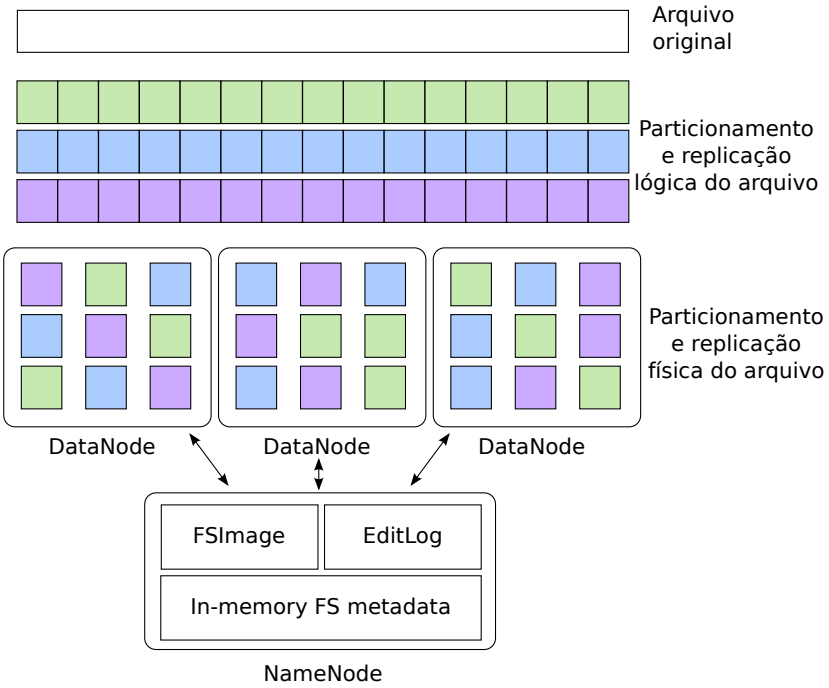

Figura 2.3. Visão geral da organização do HDFS, ilustrando o particionamento e replicação distribuída de um arquivo.

O HDFS é responsável por armazenar meta-dados sobre o sistema de arquivo bem como arquivos de usuários. Arquivos de meta-dados são armazenados em um servidor dedicado, chamado de *NameNode*, enquanto os arquivos de usuários são armazenados em outros servidores chamados de *DataNodes*. Todos os arquivos têm seus dados divididos em blocos grandes, que são distribuídos pelos discos dos *DataNodes*. Cada bloco é também replicado em outros *DataNodes* por motivos de confiabilidade na recuperação da informação. Além de garantir durabilidade dos dados, esta estratégia tem a vantagem adicional de que a largura de banda para transferência de dados é ampliada, e há mais oportunidades para escalonar a computação para perto dos dados necessários. A figura 2.3 ilustra como um arquivo é particionado em blocos de tama-

[1] As primeiras versões do HDFS não permitiam a operação de *append*, porém essa funcionalidade foi adicionada em versões posteriores (versão `0.20-append` ou `0.20.2`)

nhos iguais (geralmente 64 ou 128 MB), de maneira que cada bloco é replicado nas diferentes máquinas que compõem o cluster.

Todos os servidores são totalmente conectados e se comunicam uns com os outros por meio de protocolos baseados em TCP. Para realizar operações de leitura ou escrita de dados, um cliente requisita ao NameNode quais DataNodes devem ser acessados e, em seguida, interage diretamente com cada DataNode, nunca realizando tais operações por intermédio do NameNode.

Os predecessores do Watershed-ng, isto é, a plataforma Anthill e a versão original do Watershed, não possuiam soluções para armazenamento distribuído, de modo que o usuário era responsável por desenvolver, para cada aplicação, o código necessário para acessar a coleção de dados no sistema de arquivos locais ou arquivos remotos. O Watershed-ng é integrado com o HDFS oferecendo de maneira transparente a manipulação de arquivos distribuídos, contribuindo para necessidades das aplicações e também para tarefas internas da própria plataforma.

2.1.3 Serviço de coordenação distribuída

Aplicações distribuídas para processamento de dados massivos requerem diferentes tipos de coordenação distribuída, tais como: configurações estáticas e dinâmicas de parâmetros operacionais; eleição de lider; definição de grupos e seus elementos, geralmente definindo atributos e disponibilidade de cada elemento; implementação de acesso mutuamente exclusivo por meio de *locks* distribuídos.

ZooKeeper [Hunt et al., 2010] foi baseado principalmente no Chubby [Burrows, 2006], que é um serviço para coordenação de processos de aplicações distribuídas, oferecendo uma abstração de um conjunto de nós de dados, chamados *znodes*, organizados de acordo com um espaço de nomes hierárquico. Espaços de nomes hierárquicos são estruturas de árvores comumente usadas em sistemas de arquivos. O ZooKeeper implementa um mecanismo de monitoramento que permite aos clientes receberem notificações de eventos que acontecem em um determinado caminho do espaço de nomes. Os monitores de znodes são elementos que produzem um evento de notificação uma única vez dentro de uma sessão, isto é, os monitores são desativados quando algum evento é disparado ou com o término da sessão. Os monitores indicam apenas a ocorrência de uma alteração, especificando o tipo de operação realizada, porém não detalham especificamente qual foi a alteração realizada. A aplicação cliente é quem deve contactar o sistema para obter os detalhes desejados sobre o evento. Esse mecanismo de monitoramento pode ser utilizado, por exemplo, para realizar cache de dados, sem a necessidade do gerenciamento do cache do cliente diretamente.

Assim como os sistemas de arquivos, para se referenciar a um dado znode, é utilizado a notação padrão para caminho de arquivos em sistemas UNIX. Diferentemente dos sistemas de arquivos, todos os znodes armazenam dados e podem ter filhos, a menos dos nós efêmeros que não podem ter filhos. Existem dois tipos de znodes que podem ser criados pelo usuário do ZooKeeper:

- Regulares: usuários manipulam os znodes regulares criando, modificando e excluindo de maneira explícita.
- Efêmeros: usuários são capazes de criar znodes efêmeros, e podem explicitamente excluir tais znodes, ou o próprio sistema do ZooKeeper é capaz de excluir tais znodes caso a sessão do usuário termine, deliberadamente ou por alguma falha.

O serviço do ZooKeeper compreende um conjunto de servidores que utilizam de replicação para obter alta disponibilidade e desempenho. Ele é capaz de garantir tanto a ordenação das requisições dos clientes no modelo FIFO (*first-in first-out*) para todas as operações, quanto a linearização das escritas, implementando um protocolo, chamado Zab [Reed & Junqueira, 2008], de *broadcast* atômico baseado em líder.

O Watershed-ng utiliza do ZooKeeper para sincronizar e coordenar a execução das instâncias de cada filtro de processamento de uma aplicação Watershed. Essa coordenação consiste basicamente em iniciar a execução de cada instância dos filtros, monitorar e gerenciar o término de cada fluxo de dados, bem como o término de cada instância dos filtros.

2.2 Trabalhos relacionados

MapReduce [Dean & Ghemawat, 2008] foi um dos primeiros trabalhos de grande relevância que desenvolveu uma abstração de mais alto nível, facilitando o desenvolvimento de aplicações distribuídas mesmo para usuários com pouca experiência em paralelismo. O principal objetivo do modelo de programação MapReduce se resume em ser capaz de oferecer alto grau de paralelismo e eficiência enquanto mantem a abstração de programação de alto nível para o usuário da plataforma. Devido suas limitações de processamento em apenas duas fases, a abstração MapReduce foi posteriormente estendida em diversos trabalhos, permitindo a criação de aplicações com topologias mais complexas, como HaLoop [Bu et al., 2010] e FlumeJava [Chambers et al., 2010].

Anthill [Ferreira et al., 2005], a plataforma predecessora ao Watershed, é um sistema de processamento distribuído que implementa a abstração filtro-fluxo. Originalmente desenvolvida para estender a plataforma *DataCutter* [Beynon et al., 2000],

o modelo de programação da plataforma Anthill usa a abordagem de fluxo de dados, conhecida como *data-flow*, para decompor a aplicação em um conjunto de unidades de processamento, referenciada como filtros. Dados são transferidos pelos filtros por meio dos fluxos de dados, permitindo que dados não tipados e de tamanho fixo sejam transferidos entre os filtros. A plataforma do Anthill possui um ambiente de tempo de execução que é capaz de executar aplicações com topologia estática para o processamento de fluxos de dados. Essa plataforma é adequada para diversas aplicações de mineração de dados [Veloso et al., 2004; Ferreira et al., 2005], explorando o máximo de paralelismo das aplicações pelo uso de paralelismo de tarefas, paralelismo de dados e assincronismo.

O Watershed [de Souza Ramos et al., 2011] foi desenvolvido como sucessor do Anthill, tendo como foco principal o processamento de fluxos contínuos de dados, similar aos sistemas Storm [Toshniwal et al., 2014], S4 [Neumeyer et al., 2010], MillWheel [Akidau et al., 2013], etc. Além disso, o Watershed permite a manipulação dinâmica da topologia das aplicações, sendo possível inserir e remover filtros de processamento em tempo de execução.

Dryad [Isard et al., 2007] é um sistema que permite o desenvolvimento de aplicações distribuídas com topologias complexas, consistindo de elementos de processamentos e canais de comunicação, compondo um grafo acíclico de processamento. Dryad, assim com o Anthill e o Watershed, buscam oferecer uma interface de desenvolvimento de alto nível que possibilita especificar individualmente cada elemento de processamento, mas também como tais elementos se comunicam. Os tipos de fluxos de dados oferecidos pelo Dryad são: persistência em arquivos, canais TCP, memória compartilhada baseado em FIFO (*first-in, first-out*).

MillWheel [Akidau et al., 2013] é um modelo de programação desenvolvido especificamente para o processamento com baixa latência de fluxos de dados. Usuários escrevem a lógica de uma aplicação como nós individuais em um grafo direcionado de computação, para que eles possam definir uma topologia arbitrária e dinâmica. Coletivamente, um *pipeline* de computações dos usuários compõe um grafo de fluxo de dados, onde os registros são entregues de forma contínua ao longo das arestas do grafo. Os usuários podem adicionar e remover nós de computação na topologia de forma dinâmica, sem a necessidade de reiniciar o sistema completamente. Esse conceito é muito semelhante ao princípio de funcionamento do Watershed original, onde os filtros são os nós de computação em uma topologia dinâmica.

S4 (*Simple Scalable Streaming System*) [Neumeyer et al., 2010] é um motor distribuído para o processamento de fluxo de dados inspirado no modelo MapReduce. O projeto do S4 compartilha diversas características com o SPC (Stream Processing

Core) [Amini et al., 2006]. Ambos os sistemas são projetados para o processamento de dados massivos e são capazes de minerar informações a partir de fluxos de dados contínuos, utilizando operadores definidos pelo usuário. As principais diferenças estão no projeto arquitetural. Embora a concepção do SPC seja derivada de um modelo de assinatura, o S4 é derivado a partir de uma combinação entre o modelo do MapReduce e o modelo de Atores [Agha, 1990; Agha et al., 1997]. Seu fluxo de dados é composto apenas por pares chave-valor e o único canal de comunicação com suporte é para fluxos rotulados, onde um elemento de processamento específico é criada para cada atributo de chave.

Storm [Toshniwal et al., 2014] também é uma plataforma para o processamento de fluxos de dados, onde cada aplicação é representada por uma topologia separada. Uma topologia é um grafo orientado, podendo conter ciclos, onde os vértices representam os elementos de computação e as arestas representam fluxos de dados entre os componentes de computação.

Outra plataforma que tem ganhado popularidade recentemente é o ambiente de computação em *cluster* chamado Spark [Zaharia et al., 2010]. Esse sistema permite ao usuário combinar, de maneira transparente, processamento em *batch*, processamento de fluxos de dados e consultas interativas. O Spark é construído em torno do conceito de RDDs (*Resilient Distributed Datasets*), abstrações que implicitamente definem o padrão de comunicação entre as partições de computação. O Watershed possui diversos objetivos semelhantes aos RDDs, buscando abstrair a comunicação distribuída. No entanto, por oferecer uma abstração dos canais de comunicação como objetos de primeira classe, o Watershed também permite que o programador mais experiente defina padrões de comunicação específicos.

Nesse sentido, a ênfase em tornar os elementos de comunicação (fluxos de dados) mais flexíveis, que impulsionaram esse esforço de reengenharia, pode ser relacionado ao Coflow [Chowdhury & Stoica, 2012], uma abstração de redes de computadores para expressar as necessidades de comunicação de dados predominantes aos paradigmas de programação paralela. Coflow torna mais fácil para que as aplicações especifiquem sua semântica de comunicação para a rede, que por sua vez permite que a rede realize melhores otimizações aos padrões de comunicação mais comuns. De maneira semelhante à que o conceito de padrão de comunicação é oferecido pelo Coflow, no modelo do Watershed os usuários são capazes de especificar qual será o padrão de comunicação, mas eles também estão autorizados a desenvolver os seus próprios canais de comunicação.

Capítulo 3

Watershed-ng

Neste capítulo descrevemos a abstração do modelo de programação do Watershed-ng, bem como de seu predecessor Watershed. Nas seções seguintes apresentamos os conceitos de filtro, fluxos de dados e os principais padrões de comunicação oferecidos pela plataforma do Watershed-ng, descrevendo também como filtros e fluxos de dados se integram para compor uma aplicação em Watershed. Por fim, detalhamos os modos de operações baseados em fluxos de dados finitos (aplicações *batch*) e fluxos contínuos de dados, e o mecanismo de terminação utilizado pelo modo de operação em *batch*.

Em Watershed, uma aplicação é definida como um conjunto de filtros conectados por fluxos de dados. Dados fluem através de fluxos e são processados por filtros, potencialmente produzindo novos dados. Paralelismo é implicitamente declarado pois todos os filtros podem ser executados concorrentemente, desde que haja dados para serem processados, e cada filtro pode ser composto por múltiplas instâncias executando em paralelo. Dados de um fluxo são distribuídos entre as instâncias de acordo com a política de particionamento do fluxo e cada unidade de dados é processada independentemente das demais.

Em Watershed-ng, filtros e fluxos são objetos que podem ser definidos pelo usuário. Como mencionado anteriormente, a versão original do Watershed permitia ao programador definir o processamento realizado apenas pelos filtros e escolher a política de cada fluxo a partir de um conjunto de opções pré-definidas. Um dos objetivos do processo de re-engenharia era transformar os fluxos em abstrações de primeira classe, que podem ser extendidos pelos usuários. Essa funcionalidade pode ser atingida desenvolvendo explicitamente as ações do fluxo de dados, ou combinando diferentes elementos previamente existentes. Tais elementos e suas interfaces são discutidas a seguir.

3.1 Filtros

Na abstração do Watershed, um elemento de processamento é representado como um filtro, contendo um conjunto de portas de entrada e um conjunto de portas de saída, como ilustrado pela figura 3.1. Podemos adicionar um canal de comunicação a cada porta de entrada e um ou mais canais de comunicação a cada porta de saída. Quando um filtro deve consumir os dados produzidos por outro filtro, um canal de comunicação vincula-os dinamicamente a partir de suas portas de entrada e saída, respectivamente.

Figura 3.1. Abstração de um elemento de processamento em Watershed.

Cada filtro pode ter múltiplas instâncias de execução, o que pode ser definido pelo usuário. As instâncias de um filtro recebem os dados a partir das portas de entrada, através de uma interface baseada em eventos. Depois de um filtro processar os dados, ele pode enviar os dados transformados através de suas portas de saída, desencadeando novos eventos de entrega de dados aos filtros ligados à outra extremidade do fluxo associado àquela porta de saída.

Pode-se dizer que um filtro age como produtor quando o mesmo produz um fluxo de saída de dados que serve de entrada para um segundo filtro. De maneira semelhante, pode-se dizer que um filtro age como consumidor quando o mesmo consome um fluxo de entrada de dados que é produzido como saída por um segundo filtro.

Os principais eventos que devem ser tratados, de acordo com a aplicação, por cada filtro são:

- O evento *process* é gerado por um dos canais de entrada de fluxo de dados, sempre que houver um novo dado para ser processado por uma dada instância do filtro. O filtro deve produzir explicitamente sua saída de dados através dos canais de saída. Considerando o aspecto concorrente, variáveis mutáveis compartilhadas devem ser manipuladas de maneira segura para o uso de *threads*.

- O evento *onInputHalt* é gerado concorrentemente quando um dos fluxos de entrada terminar, propagando o sinal de fim de fluxo de dados.

3.2 Fluxos de dados

Os filtros de processamento são conectados por canais de comunicação que criam a abstração de fluxos de dados. Um fluxo consiste basicamente de um conjunto de componentes que inclui: um *sender*, um *deliverer*, uma lista de *encoders*, e uma lista de *decoders*, como ilustrado na Figura 3.2.

Nota-se pela figura que não existe um objeto que represente propriamente dito o fluxo de dados: um fluxo é basicamente uma combinação de diversos elementos que implementam, em conjunto, o canal de comunicação.

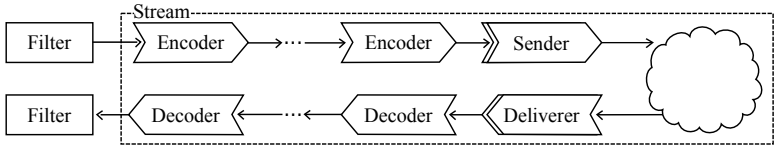

Figura 3.2. Representação da abstração de um fluxo de dados conectando dois filtros.

Senders e *deliverers* implementam a comunicação de dados real, possivelmente cruzando diferentes domínios de execução. Alguns exemplos de possíveis implementações seriam comunicação usando um sistema de arquivos distribuídos, uma conexão TCP, mensagens MPI, memória compartilhada (se a origem e o destino se encontram em uma mesma máquina), ou até mesmo por passagem de parâmetros em chamada de métodos, caso a origem e o destino sejam objetos dentro de um mesmo domínio de execução.

O *sender* deve definir um método *send*, que é chamado sempre que houver dados a serem enviados pelo canal de comunicação. O protocolo de comunicação é implementado dentro desse objeto. Por outro lado, o *deliverer* é responsável por extrair dados a partir do meio de transmissão especifico. O *sender* e o *deliverer* devem concordar sobre o meio de transmissão e o protocolo de comunicação. Quando novos dados são recebidos, o *deliverer* deve ativar um evento no próximo objeto da cadeia de transmissão (um *decoder* ou um filtro) para que os dados sejam transferidos e processados.

Os *encoders* e *decoders* operam no fluxo de dados entre os filtros e os *senders/-deliverers*. Eles podem ser utilizados como pares de elementos de transformação dos dados durante a transmissão através de um canal de fluxo de dados, por exemplo, para criptografia, compressão de dados, agregação e desagregação de mensagens, etc. Nesse caso, *encoders* e *decoders* devem ser definidos concordando na sequência de execução, de maneira que o decodificador do último codificador será o primeiro a ser executado, e vice-versa. Em suma, devem ser tratados como uma pilha de operadores.

A figura 3.3 apresentam as relações entre os diversos objetos que podem estar envolvidos na composição do fluxo de processamento de dados.

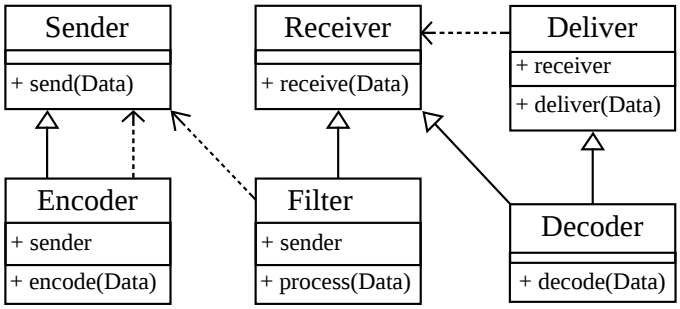

Figura 3.3. Diagrama de classes simplificado detalhando os componentes da abstração de filtros e fluxos de dados.

Podemos utilizar também *encoders* ou *decoders* isolados, como uma transformação de sentido único, *e.g.*, um seletor que encaminha apenas elementos de dados que casam com um dado padrão, um conversor de registros de fluxos de dados para o formato de JSON, ou um conversor de um fluxo de temperatura de Fahrenheit para Celsius.

A principal diferença entre o método *encode* e o método *process* pertencente à interface da classe *Filter*, é que o *Encoder* pertence a um dos canais de saída de dados do filtro, havendo portanto apenas um fluxo de saída de dados, que seria o próximo *Sender* na pilha de *senders* de saída.

Cada objeto que pode ser combinado em um canal de fluxo de dados, além de prover um método para processar os dados, deve processar também eventos gerados pelo encerramento do canal ou pelo término do fluxo de dados. O evento *onProducerHalt* é gerado quando todos os produtores de um dado *Deliverer* conectado ao fluxo de dados finalizaram suas execuções.

3.3 Padrões de comunicação

As plataformas distribuídas apresentam alguns padrões de comunicação que são prevalentes em aplicações de processamento de dados massivos [Chowdhury & Stoica, 2012]. O Watershed-ng possui uma biblioteca padrão de fluxos de dados que implementam os principais padrões de comunicação, como apresentado por Chowdhury & Stoica [2012], incluindo aqueles padrões oferecidos pelo Anthill e pela versão original do Watershed [Ferreira et al., 2005; de Souza Ramos et al., 2011].

Os fluxos de dados em rede usando TCP são:

Broadcast. Todas as mensagens de um filtro produtor são enviadas em réplicas para todas as instâncias do filtro consumidor.

Round-robin. As mensagens de um filtro produtor são enviadas sem replicação para as instância do filtro consumidor seguindo um escalonamento de *round-robin* [Nagle, 1988; Shreedhar & Varghese, 1995].

Labeled. Todas as mensagens produzidas são rotuladas, isto é, são compostas por chave e valor. Cada mensagem rotulada tem sua chave processada por uma função de *hashing* que é utilizada para selecionar a instância do filtro consumidor que receberá a mensagem rotulada [Ferreira et al., 2005; de Souza Ramos et al., 2011]. Esse padrão de comunicação também é conhecido como *shuffle* [Chowdhury & Stoica, 2012].

Os fluxos de dados em HDFS oferecidos pela biblioteca padrão do Watershed-ng implementam as funcionalidades de escrita e leitura linha-a-linha de arquivos HDFS. O fluxo de leitura por linhas em HDFS é equivalente ao LineReader implementado por um RecordReader[1] no modelo de programação da plataforma Hadoop MapReduce [White, 2009]. Os canais de comunicação por HDFS oferecem um serviço de armazenamento distribuído de maneira transparente para as aplicações em Watershed-ng, facilitando o processo de desenvolvimento de aplicações distribuídas que requerem entrada e saída de dados em arquivos com grandes volumes de dados.

A biblioteca de canais de comunicação também oferece escrita e leitura linha-a-linha de arquivos locais. Cada instância de filtro conectada à um desses canais de comunicação realizam operações em arquivos armazenados no sistema de disco local do sistema em que a instância está sendo executada.

A fim de oferecer um conjunto de operadores comuns de fluxos de dados, a biblioteca padrão do Watershed-ng também oferece alguns *encoders* e *decoders*, tais como um compressor de dados, decompressor de dados, um filtro baseado em expressões regulares, etc.

3.4 Modos de operação

Plataformas de processamento de dados massivos tais como MapReduce e Anthill, apresentam um modelo de programação com foco principal em aplicações *batch*. Apli-

[1] RecordReader é a abstração de entrada de dados no modelo de programação do Hadoop MapReduce.

cações *batch* são baseadas em fluxos de dados finitos, isto é, fluxos que têm duração bem definida e pré-determinada, tal como um fluxo de leitura de arquivo.

Além das plataformas com foco em aplicações em *batch*, existem também plataformas de processamento de fluxos massivos de dados que têm como foco principal o processamento de aplicações de fluxos contínuos de dados, geralmente se tratando de aplicações de processamento de dados em tempo real. Fluxos contínuos possuem duração indeterminada, dependendo de um ator externo para terminar sua execução, por exemplo, fluxo de dados conectado a um portal *Web*, fluxo de leitura de comentários no *Twitter*, etc.

Ambos os modelos de aplicações são de grande importância para a comunidade acadêmica e industrial. Alguns exemplos de aplicações em *batch* são soluções paralelas para algoritmos de mineração de padrões frequentes usando o modelo MapReduce [Li et al., 2008] ou o modelo filtro-fluxo do DataCutter [Veloso et al., 2004], soluções paralelas do algoritmo de PageRank usando MapReduce [Page et al., 1999; Bahmani et al., 2011], bem como algoritmos de classificação em Anthill Ferreira et al. [2005] e também MapReuce [Wu et al., 2009]. Aplicações de processamento de fluxos contínuos em tempo real incluem processamento de propagandas *on-line*, análise de sentimentos em comentários no *Twitter* [Qian et al., 2013], contagem de seguidores [Toshniwal et al., 2014] e contagem de *hashtags* mais frequentes [de Souza Ramos et al., 2011].

Enquanto a plataforma Anthill foi desenvolvida com o objetivo de oferecer uma ambiente para o processamento principalmente de aplicações em *batch*, o Watershed foi originalmente desenvolvido com o foco em aplicações de fluxos contínuos de dados. Um dos principais objetivos para o desenvolvimento do Watershed-ng é realizar uma re-engenharia do Watershed, oferecendo suporte para ambos os modos de operação, isto é, oferecendo suporte a aplicações *batch* e aplicações de fluxos contínuos de dados, convergindo o desenvolvimento das plataformas Anthill e Watershed em uma única plataforma.

3.5 Mecanismo de terminação

Em seu projeto original, o Watershed considerava todos os fluxos como contínuos, desconsiderando qualquer semântica explícita de terminação. No processo de re-engenharia da plataforma, desenvolvemos um mecanismo de terminação baseado no mecanismo utilizado na plataforma do Pregel [Malewicz et al., 2010]. A terminação de um módulo é baseada no voto pelo término de cada instância desse mesmo módulo. Um filtro possui um evento chamado *onInputHalt* que é gerado quando um dado fluxo

de entrada termina, e outro evento chamado *onAllInputsHalt* que é gerado automaticamente quando todos os fluxos de entrada terminam. Ambas as classes *Filter* e *Deliverer* possuem um método chamado *halt*. Quando o Deliverer de um fluxo executa o método *halt*, o evento *onInputHalt* é disparado na instância do filtro conectada à saída do *Deliverer*. Quando um filtro executa o método *halt*, um sinal de término é enviado ao JobMaster especificando que aquela determinada instância votou pelo término da execução. Quando todas as instâncias de um mesmo filtro votam pela terminação, o JobMaster remove esse filtro do ambiente de execução.

O mecanismo para propagação do sinal de término de um filtro pelo canal de dados, entre *sender* e *deliverer*, é de responsabilidade do protocolo de comunicação utilizado para implementar o canal de comunicação em questão, onde o *deliverer* associado ao canal deve disparar o evento *onInputHalt*. Consumidores podem decidir então terminar quando todos seus produtores de dados terminaram, com base na propagação do sinal de final de fluxo de dados.

Com esse mecanismo de terminação, é possível implementar aplicações de processamento em *batch* utilizando a plataforma do Watershed-ng, como ilustrado na seção 5. De certo modo, a implementação de uma aplicação Watershed-ng em *batch* se torna equivalente a uma aplicação Anthill [Ferreira et al., 2005], o que era exatamente um dos objetivos originais desse trabalho.

3.6 Exemplo de programação em Watershed-ng

Para ilustrar como a API (*Application Programming Interface*) do Watershed-ng pode ser utilizada para implementar uma aplicação simples, mostramos como calcular a frequência das palavras em grandes conjuntos de dados textuais.

3.6.1 Arquitetura da aplicação de exemplo

Contagem de palavras é uma aplicação de processamento em *batch* para grandes conjuntos de dados que é altamente adequado para a abstração MapReduce [Dean & Ghemawat, 2008]. A figura 3.4 apresenta uma implementação em Watershed-ng para a aplicação de contagem de palavras que é conceitualmente equivalente à mesma descrita para a abstração do MapReduce. A aplicação consiste de dois filtros: o WordCounter e o Adder[2]. A comunicação entre os filtros WordCounter e Adder usa o padrão de comunicação *labeled*, onde pares de chave-valor são entregues de acordo com o resultado de

[2] Considerando o modelo de programação MapReduce, o filtro WordCounter é equivalente ao Mapper e o filtro Adder equivalente ao Reducer.

uma função de mapeamento (basicamente uma função *hash*) usando a chave de cada par enviado, que nesse caso será cada palavra encontrada no texto de entreda.

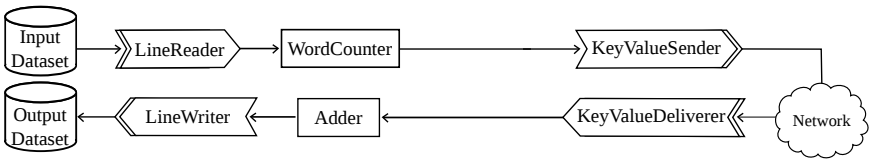

Figura 3.4. Arquitetura da aplicação de contagem de palavras no modelo de programação do Watershed-ng.

O LineReader implementa um fluxo de leitura de dados de um arquivo, realizando a leitura linha por linha. Em seguida os dados são propagados para as instâncias do filtro WordCounter. O WordCounter recebe cada linha do arquivo que é em seguida separada em uma lista de palavras. Para cada palavra encontrada em uma linha da entrada, o filtro produz pares contendo $< palavra, 1 >$. A comunicação entre os filtros WordCounter e Adder é realizada pelo padrão de comunicação *labeled* (baseado em pares chave-valor), implementado em conjunto por KeyValueSender e KeyValueDeliverer. Isso garante que todos os pares produzidos com um dada palavra serão encaminhados para uma mesma instância do filtro consumidor, independentemente de onde os pares foram produzidos, ou de quantas instâncias existam de cada filtro. O filtro Adder recebe pares na forma $< palavra, frequência >$, realiza a soma total das frequências locais de cada palavra recebida, produzindo pares únicos $< palavra, frequência >$ com a frequência global para cada palavra.

3.6.2 Implementação da aplicação de exemplo

Os filtros WordCounter e Adder estão apresentados nas figuras **??** e 3.5, respectivamente.

Na figura **??**, vemos que o WordCounter possui apenas um método de interesse, o método `process`. Esse método é ativado pelo ambiente de execução do Watershed sempre que houver novos dados para serem processados. Os parâmetros identificam o fluxo de entrada de dados (`src`) e o objeto com o dado recebido (`obj`). Para enviar uma mensagem através de um fluxo de dados de saída basta utilizar o método `send` no canal de comunicação.

O filtro Adder, ilustrado na figura 3.5, é um pouco mais complexo. A fim de gerar um único par para cada palavra, é preciso manter um dicionário local para armazenar a contagem parcial por palavra. Para isso, é definido um método `start` e um `finish` para construir tais estruturas de dados e produzir a contagem final no fluxo de saída.

```
package sample.wordcount;

import java.util.Map;
import java.util.Map.Entry;
import java.util.concurrent.ConcurrentHashMap;

import hws.core.Filter;

public class Adder extends Filter{
   private Map<String, Integer> counts;

   public void start(){
      super.start();
      counts = new ConcurrentHashMap<String, Integer>();
   }

   public void finish(){
      for(Entry<String,Integer> entry: counts.entrySet())
         outputChannel().send(entry.toString());
      super.finish();
   }

   public void process(String src, Object obj){
     Entry<String, Integer> data = (Entry<String, Integer>)obj;
     String word = data.getKey();
     Integer val = data.getValue();
     if(counts.containsKey(word))
        counts.put(word, val+counts.get(word));
     else counts.put(word, val);
   }
}
```

Figura 3.5. Código fonte para o filtro Adder da aplicação de contagem de palavras usando a plataforma Watershed-ng.

3.6.3 Configuração da aplicação de exemplo

Cada filtro deve ser configurado separadamente para identificar suas conexões, definir o número de instâncias, além de outras informações. A figura 3.6 apresenta o arquivo de configuração para o filtro Adder, utilizando o sistema de arquivo local no canal de saída de dados. Primeiramente, o filtro é identificado incluindo a classe que o implementa e o arquivo `jar` que contém tal implementação. Em seguida, as portas de entrada e saída são listadas. Para essas portas, o atributo `name` define os nomes dos fluxos que serão acessíveis pelo filtro (por exemplo, utilizando o método `outputChannels()`). No caso do filtro Adder, o fluxo de entrada é um canal de comunicação via rede utilizando o padrão *labeled*, dadas as classes definidas para o *sender* e *deliverer* (`KeyValue`). A porta de saída está ligada a um fluxo de saída direcionado para um arquivo no sistema

```
<?xml version="1.0"?>
<filter name="adder" file="sample.jar"
   class="sample.wordcount.Adder" instances="2">
    <input>
        <channel name="net">
            <sender class="hws.channel.net.KeyValueSender" />
            <deliver class="hws.channel.net.KeyValueDeliver" />
            <encoders>
                <encoder class="sample.wordcount.Combiner"
                   file="sample.jar" />
            </encoders>
        </channel>
    </input>
    <output>
        <channel name="writer">
            <sender class="hws.channel.lfs.LineWriter" >
                <attr name="path" value="/watershed/wordcount" />
            </sender>
        </channel>
    </output>
</filter>
```

Figura 3.6. Arquivo de configuração para o filtro Adder usando a plataforma do Watershed-ng.

de arquivos local dos nós de computação escalonados para executar as instâncias do filtro.

3.6.4 Otimização utilizando um agregador local

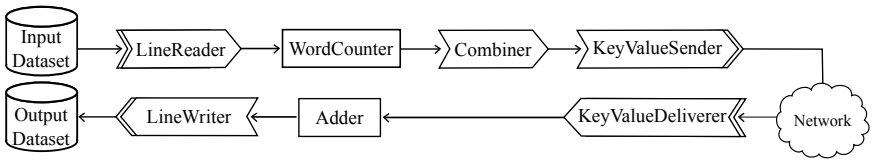

Figura 3.7. Arquitetura da aplicação de contagem de palavras no modelo de programação do Watershed-ng, incluindo o elemento de agregação local.

Uma otimização comum realizada na aplicação de contagem de palavras consiste em agregar a frequência parcial de cada palavra, realizada localmente em cada instância do filtro WordCounter, reduzindo a comunicação entre as instâncias dos filtros WordCounter e Adder. Dessa maneira, cada instância do filtro WordCounter produz uma única saída $< palavra, frequência >$ para cada palavra encontrada. Esse elemento de agregação local, conhecido como *Combiner*, é encontrado no modelo de programação MapReduce [White, 2009].

Considerando a abstração de fluxo de dados em Watershed-ng, podemos pensar no agregador local como uma transformação no fluxo de dados, que pode ser implementada por um *encoder*, como ilustrado na figura 3.7. O código do agregador é estruturado de maneira similar ao código do Adder (figura 3.5).

Capítulo 4

A arquitetura do Watershed-ng

Em sua implementação original, o framework do Watershed foi implementado de maneira monolítica em um sistema único. Todos os elementos de execução, tais como alocação de recurso, escalonamento, tolerância à falhas e interfaces de I/O, eram controladas internamente, o que resultava em um sistema de tempo de execução muito complexo, limitando também a gama de técnicas disponíveis que poderiam ser implementadas pela equipe local.

Aquela versão do Watershed utilizava escalonamento estático: o usuário atribuía as instâncias de cada filtro a máquinas específicas no arquivo de configuração do aplicativo. O controle de acesso e a execução das tarefas foram implementados usando comandos MPI, considerando que MPI também foi usado como substrato de comunicação. Além disso, não havia controle de alocação; diferentes usuários compartilhando o cluster, para executar aplicações simultaneamente, poderiam facilmente degradar o desempenho conjunto simplesmente por escolherem, de maneira não intencional, um mesmo nó para executar instâncias de filtros.

Visando evitar tais problemas, decidimos utilizar o YARN como escalonador e alocador de recrusos para a nova implementação do sistema. Sendo o negociador e escalonador de tarefas do ambiente Hadoop, o YARN oferece uma plataforma base para o gerenciamento de aplicações distribuídas em uma abstração de alto-nível. Ao fazer isso, ele ainda permite que aplicações de diferentes plataformas coexistam em um mesmo ambiente de cluster.

Recentemente, muitas plataformas, como Spark [Zaharia et al., 2010] e Storm [Toshniwal et al., 2014], por exemplo, começaram a tirar proveito de funcionalidades do ecossistema Apache Hadoop [Murthy et al., 2013] para simplificar suas implementações. A versão atual foi organizada em módulos de forma que, além de oferecer o modelo de programação MapReduce, seus módulos oferecem funcionalidades

úteis que podem ser integradas a outras aplicações [Vavilapalli et al., 2013].

Em particular, para o Watershed-ng, utilizamos o YARN como escalonador de processos e gerenciador de recursos, o Zookeeper para implementar o serviço de coordenação para os múltiplos processos de cada aplicação, e o HDFS tanto como um meio de distribuição dos arquivos executáveis quanto como um sistema de arquivos distribuídos para a camada de aplicação. Além desses serviços pré-existentes, a plataforma do Watershed-ng compreende três componentes específicos: (*i*) o *JobClient*, responsável por submeter uma aplicação ao ambiente Watershed-ng; (*ii*) o *JobMaster*, que controla a execução dos processos que compõem a aplicação; e (*iii*) os *InstanceDrivers*, que executam e monitoram cada instância dos filtros. Uma visão geral da arquitetura completa está ilustrada na figura 4.1.

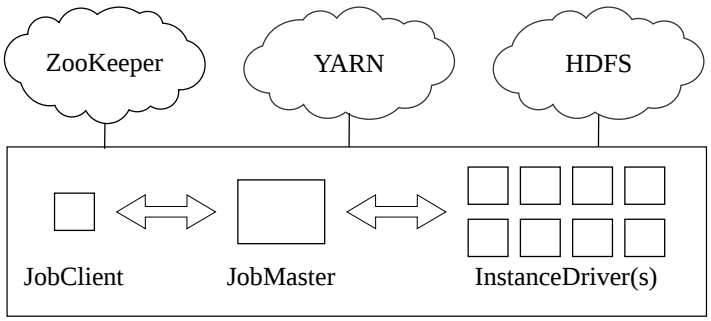

Figura 4.1. Uma visão geral em alto nível da arquitetura do Watershed-ng implementada sobre o ecossistema Hadoop.

4.1 Iniciando uma aplicação

Nesta seção nós descrevemos a sequência de etapas executadas ao iniciar a execução de uma aplicação em Watershed-ng, incluindo a interação com os módulos do ecossistema Hadoop. Essa sequência é mostrada na figura 4.2; na discussão que se segue, os números em parênteses referem-se às interações enumeradas nessa figura.

Quando um usuário deseja iniciar uma aplicação Watershed-ng, ele deve iniciar um JobClient com os arquivos de configuração que descrevem quais filtros devem ser executados e como eles se conectam (um exemplo de configuração é apresentado pela figura 3.6). Para o usuário, o JobClient serve de interface de execução de uma aplicação em Watershed-ng. Ele é responsável por analisar os arquivos de configuração (1), identificando os filtros e suas conexões por fluxos de dados, como definidos pelo usuário. Em seguida, (2) ele contata o YARN para registrar o código que executará o

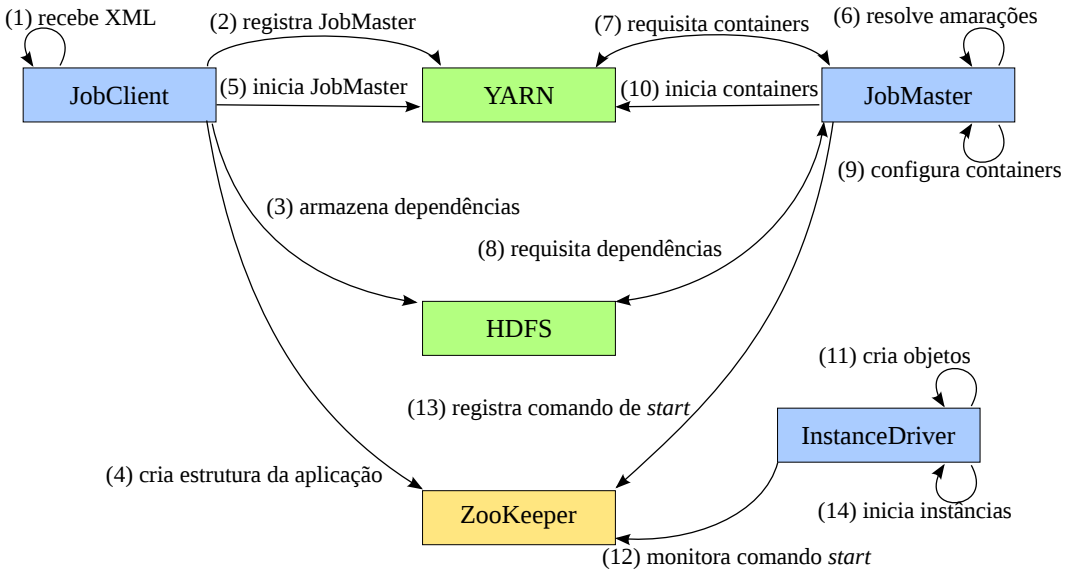

Figura 4.2. Diagrama com a sequência dos principais eventos realizados pelos componentes da plataforma Watershed-ng para iniciar uma aplicação.

JobMaster, componente principal responsável por gerenciar e escalonar as aplicações em Watershed-ng.

O JobClient também possui responsabilidades relacioanadas à gerenciamento interno de tarefas, *e.g.,* criação de uma estrutura de sub-árvore no HDFS para o espaço de nomes da aplicação (3), utilizada principalmente para compartilhar arquivos executáveis e de bibliotecas, bem como para o armazenamento de arquivos de *log*; criação de uma estrutura de sub-árvore no ZooKeeper para o espaço de nomes da aplicação à ser executada (4), necessário para a coordenação distribuida das tarefas.

Uma vez que essas configurações estão em vigor, então o JobClient contata o YARN emitindo um pedido para iniciar um processo do JobMaster (5). Em seguida, ele aguarda a resposta final do YARN Manager.

O YARN realiza a alocação de recursos para um container, onde o JobMaster é iniciado, passando os objetos identificados pelo cliente (5). O JobMaster basicamente recebe e decodifica os descritores dos módulos (6), requisita ao ResourceManager do YARN os containers necessários para executar cada instância dos filtros (7), configurando em seguida o ambiente de cada container com as bibliotecas compartilhadas relacionados à aplicação em execução (8). Durante o processo de requisição dos containers, dependendo da estratégia de escalonamento do JobMaster, um conjunto de propriedades são requisitados para cada container, por exemplo, especificação da capacidade da memória principal, número de núcleos virtuais de CPU, etc., tais propriedades são então proporcionadas pelo ResourceManager do YARN,

Para cada container, as configurações necessárias são realizadas (9) e um InstanceDriver individual é executado (10), recebendo como parâmetro o descritor da instância do filtro que deve ser iniciado pelo container.

O InstanceDriver recebe um descritor da instância de um filtro, instanciando os objetos que compõem o filtro (11), registra seu estado atual no ZooKeeper (12), e por fim, aguarda um sinal do JobMaster para iniciar a execução da instâncias do filtro dado (13). Quando o sinal de inicio é recebido, cada InstanceDriver gerencia a execução das instâncias dos filtros (14). O InstanceDriver também possui responsabilidades referente ao gerenciamento de sua coordenação, tais como a criação de um sinal de terminação de suas instâncias no ZooKeeper, geração dos eventos de *onProducersHalt* resultantes de eventos gerados por sinais via ZooKeeper.

O JobMaster observa o progresso dos executores da aplicação e quando todos os filtros estão prontos para procederem, ele envia um sinal pelo ZooKeeper. Quando todas as instâncias de um filtro terminam, ele retorna o controle ao JobClient, que termina por limpar o ambiente de execução da aplicação.

4.2 Implementação dos fluxos

A primeira versão do Watershed possuía apenas implementações pré-definidas dos fluxos, que eram implementados usando MPI e ofereciam as políticas básicas herdadas do Anthill: *broadcast*, *round-robin* and *labeled*. Não havia nenhuma API pré-definida para interagir com um sistema de arquivo: os desenvolvedores eram responsáveis por abrir, ler e escrever em arquivos diretamente no filtro, o que não era sempre simples, considerando a natureza distribuída da computação[1]. A nova versão, Watershed-ng, permite que usuários definam seus próprios componentes de fluxos, incluindo suas implementações. Para demonstrar a flexibilidade dessa solução e para proporcionar um conjunto mínimo de funcionalidades, nós implementamos as políticas originais do Anthill usando *sockets* TCP, um par de fluxos para leitura/escrita no sistema de arquivos local do nó de computação e também um par de fluxos para leitura e escrita de arquivos diretamente do HDFS.

Os fluxos de leitura/escrita locais são facilmente implementados com filas em memória que interagem com *threads* de leitura/escrita simples em arquivo. O nome dos arquivos que serão usados são informados pelos arquivos de configuração e verificados

[1] Essa foi realmente uma das razões para permitir que os usuários pudessem definir o escalonamento das instâncias explicitamente em termos de máquinas, visando garantir que um filtro que necessitava ler um arquivo armazenado no sistema de arquivos local de uma máquina específica seria posicionado adequadamente.

pelo InstanceDriver quando os fluxos são construídos. O arquivo de configuração da figura 3.6 demonstra como o filtro Adder foi configurado para escrever em um arquivo no diretório local.

Para o HDFS, nós desenvolvemos fluxos de entrada e saída para os casos onde a saída de um filtro deve armazenar dados em um arquivo ou quando um filtro precisa ler de um arquivo, respectivamente. Atualmente o Watershed oferece apenas fluxos que suportam arquivos orientados a linha. O fluxo *LineReader* particiona a entrada igualmente entre as instâncias do filtro que requisitou a leitura, considerando o tamanho em bytes de cada partição. Entretanto, nenhuma linha é dividida entre duas partições ou enviada para mais de um filtro consumidor. O fluxo *LineWriter* é uma interface simples de saída para o HDFS, onde cada dado produzido é armazenado como uma linha de um arquivo em HDFS.

4.3 Sistema de arquivo distribuído

Ao iniciar, o JobClient cria, se necessário, uma estrutura de espaço de nomes no HDFS para ser utilizado pelo Watershed-ng. Essa sub-árvore de diretórios é utilizada principalmente para compartilhar arquivos executáveis e bibliotecas de dependência. A mesma estrutura no HDFS é utilizada para o armazenamento de arquivos de *log* referentes à execução de cada aplicação.

Os arquivos executáveis e bibliotecas são organizados separadamente entre a plataforma Watershed-ng e suas aplicações. Os arquivos necessários para a execução da plataforma são armazenados em `/hws/bin`. Cada aplicação possui um sub-diretório em `/hws/apps/<app_id>`, onde `<app_id>` é obtido dinamicamente pelo JobCliente ao registrar uma nova aplicação na plataforma do YARN.

Cada container em uma aplicação Watershed-ng, incluindo o JobMaster, cria um arquivo no HDFS para registrar eventos de execução, tais como exceções que possam ocorrer. O arquivo de *log* é criado entre os sub-diretórios da aplicação, sendo `/hws/apps/<app_id>/logs/<container_id>.log` o caminho para o arquivo de *log* de um dado container.

O mecanismo de *log* segue o padrão *singleton* onde todos os elementos instanciados por um mesmo container têm acesso atômico ao mesmo arquivo HDFS para registros de eventos. O sistema de log é acessado internamente pela plataforma do Watershed-ng bem como pela camada de aplicação.

4.4 Coordenação distribuída

A coordenação distribuída de processos em execução para uma determinada aplicação é realizada por meio do ZooKeeper. O ZooKeeper é utilizado principalmente para sincronizar o processo de inicialização das instâncias dos filtros e armazenar informações de quais quais instâncias finalizaram o processamento. Além disso, ele é utilizado também para auxiliar na implementação do mecanismo de terminação, permitindo implementar um mecanismo para votação de término com consistência.

Além de ser utilizado para realizar coordenação distribuída, gerenciando a execução das instâncias dos filtros, novas funcionalidades referentes à tolerância a falhas também podem ser desenvolvidas utilizando o ZooKeeper, como discutido posteriormente.

4.5 Habilitando topologias dinâmicas

A descrição do funcionamento do Watershed-ng até aqui considera a execução de aplicações *batch*, como no Anthill. Para oferecer uma abstração semelhante ao Watershed original, onde as aplicações consistem principalmente de fluxos de dados contínuos e filtros podem ser acoplados a fluxos de forma dinâmica, descrevemos uma solução que pode ser utilizada, estendendo a plataforma atual do Watershed-ng.

Inicialmente, um SystemMaster (equivalente ao conceito de um JobMaster contínuo) é instanciado via YARN como um Application Master. Diferentemente do JobMaster, o SystemMaster não recebe os descritores da topologia de uma aplicação como parâmetro de criação. Quando iniciado, o SystemMaster cria dois znodes efêmeros (vide seção 2.1.3): o primeiro com o endereço do nó de computação local e o segundo com a porta de rede do servidor de submissão de filtros. O SystemMaster é responsável por (*i*) escalonar os filtros entre as máquinas disponíveis; (*ii*) gerenciar dinamicamente a topologia do ambiente de execução, adicionando ou removendo filtros e realizando as conexões entre os filtros em execução; (*iii*) gerar os eventos de execução para as instâncias de cada filtro em execução, tais como eventos relacionados ao término dos filtros produtores de dados para um determinado fluxo, eventos de inicialização ou término das instâncias de um filtro, etc.

A aplicação cliente permite que o usuário adicione ou remova filtros ao ambiente de execução dinamicamente. Ao adicionar um novo filtro, o SystemMaster ativo requisitaria os containers para executar cada instância do novo filtro. Após receber os containers requisitados, o SystemMaster configuraria o ambiente de execução dos containers, adicionando as dependências de execução do InstanceDriver e dos módulos

da aplicação. Por fim, o SystemMaster submeteria o InstanceDriver para executar nos containers, com os devidos parâmetros de execução.

O SystemMaster pode persistir informações da topologia do ambiente de execução em uma estrutura de znodes no ZooKeeper. Diversas instâncias do SystemMaster podem ser criadas e por um sistema de eleição será decidido qual instância será o SystemMaster ativo. As demais instâncias do SystemMaster monitoram os znodes efêmeros com a informação de que a instância atual do SystemMaster está ativa. Quando aquele znode é removido, uma nova eleição é realizada para definir a próxima instância que assumirá a posição de SystemMaster ativo. O novo SystemMaster ativo recupera as informações da topologia de execução direto do ZooKeeper e anuncia sua tomada de posse.

Com o objetivo de facilitar a interface do usuário com o ambiente de topologia dinâmica, é necessário desenvolver um mecanismo de tipagem semântica para os fluxos de dados, de modo que seja possível identificar o fluxo de dados não apenas pelo nome da porta em que o fluxo é produzido, mas também pelo seu conteúdo. Essa abordagem se assemelha à alguns conceitos de redes orientadas a conteúdo [Arianfar et al., 2010] onde a definição de uma conexão na rede se dá através da descrição dos interesses do cliente, chamada de *interests* ou *subscriptions*, ao invés de ser definida pelo endereço da máquina servidora do conteúdo. Essa nova abstração para definição de fluxos de dados seria possível, desenvolvendo componentes dos canais de comunicação que registram suas especificações em um serviço para armazenamento da tipagem semântica dos fluxos de dados. Dessa maneira, a relação entre filtros produtores e consumidores se daria através de uma descrição semântica do conteúdo do fluxo de dados em questão.

4.5.1 Adicionando filtros dinamicamente

Quando o usuário deseja adicionar um novo filtro de processamento, F_1, ele envia um descritor desse filtro para o SystemMaster. Em seguida, o SystemMaster seleciona quais nós de computação serão responsáveis pela execução de cada instância do filtro e então envia o descritor de cada instância para os nós de computação selecionados. O filtro consumidor é responsável por descrever os componentes que compõem os canais de fluxos de entrada, como representado pela figura 3.2, criando a conexão com os filtros produtores. Desse modo, diferentes consumidores são capazes de utilizar diferentes políticas de leitura de uma mesma saída de um determinado filtro produtor.

Para cada filtro F_0 que é um produtor de dados para F_1, o SystemMaster envia a lista de *encoders* e o *sender* para cada instância de F_0, que serão ligados à porta de saída apropriada, como especificado por F_1. De maneira semelhante, para cada filtro

F_2 que é um consumidor de um fluxo de dados produzido por F_1, o SystemMaster informa a lista de *encoders* e o *sender* para cada instância de F_1, como especificado pelo fluxo de entrada de F_2.

Capítulo 5

Avaliação Experimental

Neste capítulo avaliamos o Watershed-ng em termos de eficiência e tamanho do código necessario para implementar um dado algoritmo. Para as aplicações descritas a seguir, comparamos suas eficiências com outras implementações disponíveis do mesmo algoritmo, comparando o código desenvolvido em cada caso baseado no número de linhas de código (LOC) para ter uma noção da complexidade de programação em cada uma das plataformas.

Apesar das limitações conhecidas da métrica de LOC para comparação da complexidade de programação, para as amostras de aplicações como as de tamanhos consideradas por esse trabalho, onde cada amostra foi produzido por programadores experientes na plataforma, uma diferença significativa no número de linhas de código necessárias para implementar um mesmo algoritmo pode servir como um primeiro indício do poder de expressão de cada plataforma sendo analisada.

5.1 Complexidade de desenvolvimento das plataformas

A tabela 5.1 apresenta os uma comparação entre o número de linhas de código das plataformas Anthill, Watershed e Watershed-ng. O valor de LOC da plataforma Watershed-ng apresentado pela tabela representa apenas o código principal da plataforma, desconsiderando os fluxos de dados oferecidos. Os padrões de comunicação em rede possuem apenas 272 LOC, enquanto os padrões de comunicação em HDFS 211.

O Watershed-ng representa 25% do código do Anthill e 48% do Watershed original. Essa redução considerável no código da plataforma é atribuído principalmente

Plataforma	Linguagem de Programação	Linhas de Código
Anthill	C/C++	8721
Watershed	C/C++	4461
Watershed-ng	Java	2138

Tabela 5.1. Linhas de código referente as implementações de cada uma das plataformas.

pela utilização de serviços que auxiliam no desenvolvimento de aplicações distribuídas, isto é, os serviços do ecossistema Hadoop, incluindo o YARN, o HDFS e o ZooKeeper. Como podemos ver na seção seguinte, apesar do Watershed-ng possuir um código mais simples se comparado às implementações originais do Anthill e Watershed, o Watershed-ng apresenta bons desempenhos quando comparado às outras plataformas.

5.2 Aplicações

Nesta análise, consideramos duas aplicações em *batch*: conta palavras e kNN. A aplicação de contagem de palavras é uma aplicação simples e com comportamento bem definido, permitindo uma melhor análise entre as diferentes plataformas.

5.2.1 Conta palavras

O problema de contagem de palavras e sua implementação no novo ambiente de processamento do Watershed são discutidas na Seção 3.6. Para a comparação de eficiência, consideramos versões em todos os ambientes de execução com e sem o uso de elemento de combinação para reduzir o número de mensagens em trânsito.

Para o Watershed-ng, consideramos as versões usando o HDFS e também acessando o sistema local de arquivos, visando isolar esse fator durante a comparação com os demais ambientes de execução. Isso resultou em um total de nove cenários diferentes: duas versões do Anthill, com e sem o elemento de agregação local (nesse caso, a funcionalidade de agregação teve que ser implementada diretamente no filtro contador, de modo que o mesmo produza apenas a contagem agregada local); duas versões em Hadoop MapReduce, também com e sem o elemento de agregação local, isto é, um Combiner; quatro versões usando o Watershed-ng: com e sem o uso de agregação local, usando um encoder como o Combiner, e com o uso dos fluxos via HDFS ou sistema de arquivos local (LFS); e uma versão implementada em Spark, que por padrão já possui o efeito de agregação local implementado diretamente em seus operadores.

5.2.2 Classificador *k-Nearest Neighbors*

Como uma segunda aplicação *batch*, implementamos um algoritmo comum de classificação em mineração de dados, o classificador *k-nearest neighbors* (kNN). O classificador kNN infere a classe de uma dada entrada com base na classe majoritária entre seus k vizinhos mais próximos computados a partir de uma base de dados de treinamento [Zaki & Wagner Meira, 2014].

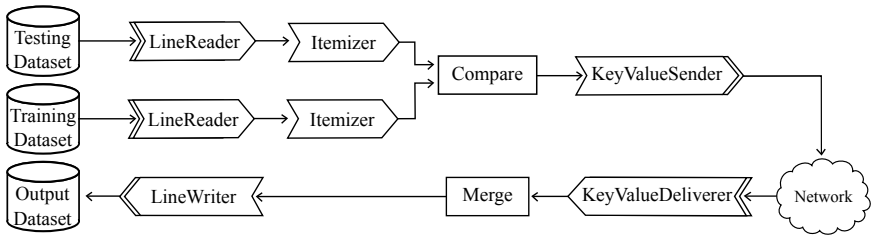

Figura 5.1. Visão geral do algoritmo kNN (*k-Nearest Neighbors*) no modelo filtro-fluxo da plataforma Watershed-ng.

A figura 5.1 ilustra a organização do algoritmo do classificador kNN usando a abstração do modelo de programação em Watershed-ng. Utilizamos os fluxos de dados com os padrões de comunicação como explicados na seção 3.3. O fluxo de entrada passa por um enumerador, implementado por um *decoder*, de modo que cada dado de entrada é propagado como um par chave-valor, com o dado de entrada sendo o valor e um identificador incremental como sendo a chave. O identificador é importante para a fase de agregação, chamada de *Merge*, onde os resultados parciais dos k vizinhos locais mais próximos, produzidos pelas instâncias do filtro *Compare*, são combinados por chave, compondo a classificação baseada nos k vizinhos globais mais próximos.

Cada instância do filtro *Compare* recebe a base de teste completa e apenas uma porção disjunta da base de treinamento. Em seguida, cada instância produz os k vizinhos mais próximos relativos à sua porção da base de treinamento. Para a comunicação Compare-Merge utilizamos o padrão de comunicação *labeled*, que implementa a comuninicação via rede baseado em pares chave-valor é utilizado para garantir que os vizinhos candidatos são apropriadamente agrupados. O filtro Merge calcula os k vizinhos globais mais próximos e produz a classificação dos dados de entrada.

A implementação do kNN em Spark utiliza o conceito de *Resilient Distributed Datasets* (RDDs) que é intrínseco ao seu modelo de programação. A base de dados de treinamento é lida para um RDD, que é particionado em diversas porções, distribuídas entre os nós de computação. Em seguida, porções de entrada de teste é replicada para cada partição com os pontos de treinamento. Dentro de cada partição, k vizinhos mais próximos são computados de maneira distribuída para cada ponto da entrada de teste.

Por fim, os vários vizinhos encontrados são combinados e processados, calculando os k vizinhos mais próximos, no geral, para cada ponto da entrada de teste, produzindo a classificação final baseado na classe majoritária.

5.3 Complexidade do código

Para esta análise, consideramos apenas as versões de código com o agregador local, considerando que foi a versão com melhor desempenho. Além do código desenvolvido para o Watershed-ng, consideramos também as implementações desenvolvidas anteriormente para o Anthill, bem como para a versão original do Watershed. Consideramos a implementação em Hadoop MapReduce apenas para a aplicação de contagem de palavras, pois é bem adequada para o modelo de programação oferecida pelo MapReduce.

Plataforma	Aplicação	
	conta palavras	kNN
Anthill	210	332
Watershed Original	127	400
Watershed-ng	65	145
Hadoop	64	NA
Spark*	18	83

Tabela 5.2. Linhas de código referente as implementações das aplicações em cada uma das plataformas. (*) A plataforma Spark utiliza um paradigma de linguagem de programação diferente das demais plataformas,

A Tabela 5.2 sumariza os resultados. Os códigos escritos para o Watershed-ng foram menores que aqueles escritos para seus antecessores (Anthill e o Watershed original). Quando a aplicação é adequada para o modelo de programação do Hadoop MapReduce, como ocorre por exemplo com a aplicação de contagem de palavras, os programas possuem tamanhos similares. A redução obtida com a nova plataforma vem principalmente do fato de que o acesso a arquivos está embutido na definição dos canais de fluxo de dados e que a nova abstração e estruturação dos canais de comunicação facilita adicionar elementos de transformação dos dados usando *encoders* e *decoders*.

A implementação em Spark requer consideravelmente menos linhas de código quando comparada com as outras plataformas. Isso se dá principalmente pela utilização das versões implementadas em Scala, que é uma linguagem funcional que tende a ser altamente expressiva, enquanto as outras plataformas são implementadas em linguagens imperativas. Entretanto, linguagem de programação imperativa ainda é o paradigma dominante em uso geral.

5.4 Avaliação de desempenho

Para a avaliação de desempenho, os experimentos foram conduzidos em um *cluster* contendo seis nós de computação, onde um dos nós foi configurado como o mestre enquanto os demais como escravos. Cada nó tinha um processador Intel Xeon® com 4 núcleos, frequência de *clock* de 2,5 GHz e 8 GB de RAM. Realizamos um total de cinco execuções para cada experimento e os resultados apresentados representam a média dos valores de cada execução.

Não apresentamos resultados do Watershed original. Como mencionado anteriormente, essa implementação foi planejada pensando no processamento de fluxos de dados contínuos e persistentes, utilizando uma implementação de fluxo com armazenamento local dos dados para acesso retroativo, o que fez o seu desempenho inaceitável para aplicações em *batch* — esta era, afinal, uma das principais razões para a re-engenharia da plataforma.

5.4.1 Conta palavras

A base de entrada utilizada representava uma porção de 5 GB de artigos em inglês da Wikipédia[1] com aproximadamente $61,5$ milhões de linhas de texto. As figuras 5.2 e 5.3 apresentam os resultados do tempo de execução, à medida que o tamanho da amostra da base de entrada era variada.

Como podemos observar, a versão do conta palavras em Anthill sem a otimização de agregação local apresentou um desempenho muito ruim e seus tempos de execução se tornaram proibitivos para as amostras com mais de 165 MB. Esse resultado pode estar relacionado ao mecanismo de comunicação implementado na versão original do Anthill, incluindo o mecanismo necessário para sincronização e coordenação durante a execução das instâncias dos filtros. Entretanto, a versão do conta palavras com a otimização apresenta melhor desempenho se comparado à versão sem otimização em Anthill. Apesar disso, quando comparado com as versões otimizadas para as outras plataformas, mesmo a versão otimizada em Anthill não se mostra muito eficiente (veja a figura 5.3).

Um ponto interessante foi o de que o uso de HDFS ou o sistema de arquivos local não foi um fator determinante para o desempenho: versões do Watershed-ng que diferem apenas sobre esse aspecto obtiveram desempenho semelhante, geralmente com a versão que utiliza o HDFS obtendo um desempenho ligeiramente melhor.

[1] Obtida no endereço `http://dumps.wikipedia.org/enwiki` em 12 de Janeiro de 2015.

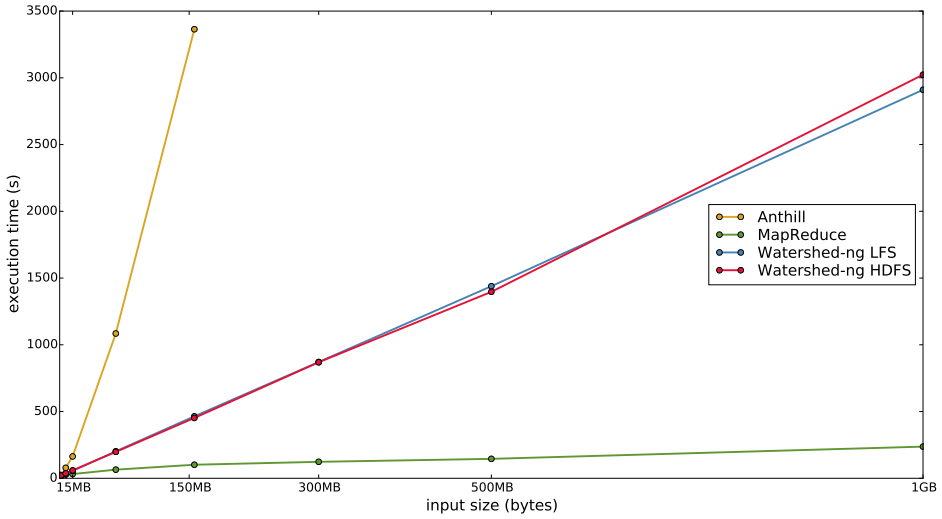

Figura 5.2. Tempo de execução da aplicação de contagem de palavras implementada em diferentes plataformas, sem utilizar a otimização de agregação local. Para comparação, utilizamos diversos tamanhos de porções da base de artigos em inglês do Wikipédia.

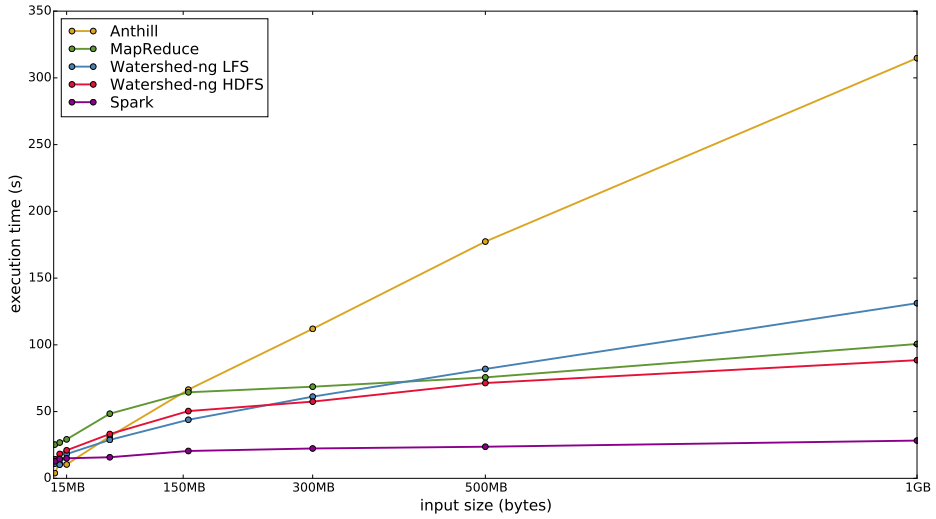

Figura 5.3. Tempo de execução da aplicação de contagem de palavras implementada em diferentes plataformas, utilizando a otimização de agregação local.

Entretanto, a versão com sistema de arquivos local exige configuração mais cuidadosa, sendo necessário que o usuário replique manualmente o arquivo de entrada em cada nó de leitura. Em contrapartida, a utilização do HDFS permite manipular transparentemente grandes arquivos de entrada de maneira distribuída. As implementações

que não usavam o elemento de agregação apresentaram um desempenho pior do que aquelas com o agregador, uma vez que o número de mensagens em trânsito era muito maior para as primeiras, onde pequenas mensagens causavam atrasos diretamente na implementação dos canais de comunicação via TCP.

A implementação em Hadoop MapReduce não foi sensível em relação ao uso do agregador local até a marca de 1 GB para o arquivo de entrada. Pelo fato da comunicação entre *mappers* e *reducers* ser implementada tendo a saída dos *mappers* armazenada em arquivos, que são posteriormente acessadas por completo pelos *reducers*, de modo que mensagens pequenas na verdade nunca atravessam separadamente a rede.

A implementação em Spark claramente obteve o melhor desempenho para essa aplicação. Isso se deve principalmente pelo seu sistema de execução leve e pelo fato de que operações funcionais geralmente podem ser melhores otimizadas, como é o caso das operações dos RDDs em Spark.

Quando consideramos uma porção de 5 GB da base de dados (não apresentado na figura 5.2), observamos que o tempo de execução para a versão em Hadoop MapReduce sem o agregador cresce significativamente (cerca de 25 minutos). Os tempos para a versão do Watershed-ng sem o agregador não foram medidos, pois tais experimentos estavam demorando mais de uma hora para terminar. A versão do Hadoop utilizando o agregador levou 336 segundos ($5,6$ minutos), enquanto a versão do Watershed-ng também utilizando agregação local levou 519 segundos ($8,7$ minutos). Nesse caso também, a utilização de arquivos intermediários contribuiu para a implementação em Hadoop MapReduce, uma vez que o agregador em Watershed reduz o número de mensagens, mas não as agrupam tanto quanto os arquivos. Para uma amostra maior, o número de pequenas mensagens na rede prejudicam o desempenho total da aplicação. (Uma maneira de corrigir esse problema de eficiência seria adicionar um par de *encoder*/*decoder* à aplicação, agrupando mensagens em blocos maiores antes de enviá-las através da rede.)

5.4.2 Classificador *k-Nearest Neighbors*

Para os experimentos do classificador kNN, utilizamos uma base de dados sintética, desenvolvida gerando pontos em um espaço bidimensional usando um conjunto de centróides, em torno dos quais foram adicionados pontos de acordo com uma distribuição normal. Os experimentos foram realizadas com um valor de k igual a 50, um total de 20000 amostras de treinamento e diferentes quantidades de amostras a serem classificadas.

A figura 5.4 apresenta os tempos de execução do classificador kNN. Para essa

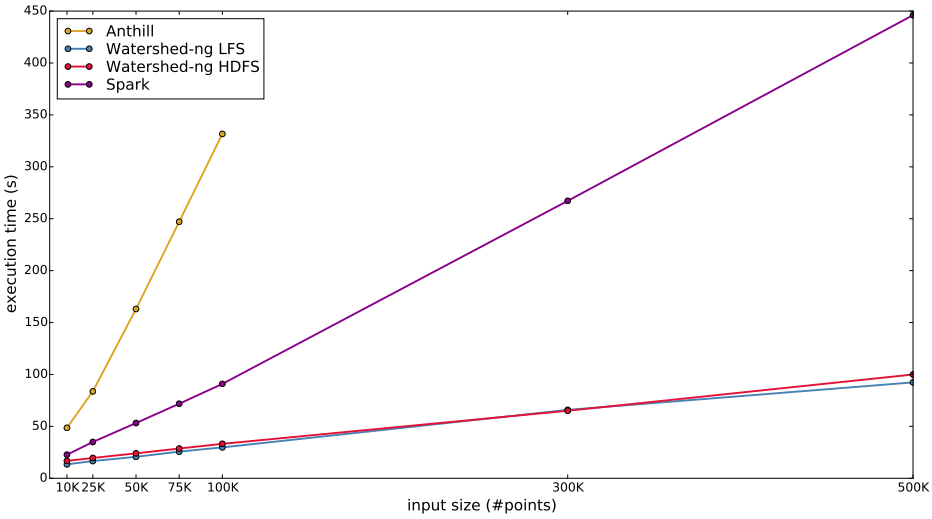

Figura 5.4. Tempo de execução da aplicação de classificação utilizando kNN (*k-Nearest Neighbors*), implementada em Anthill e Watershed-ng.

análise, não utilizamos a plataforma Hadoop MapReduce, pois o algoritmo não se adapta bem ao modelo de programação em MapReduce, resultando em um código mais complexo, maior e mais lento. Novamente, o tempo de execução do Anthill degrada rapidamente, onde para conjuntos de dados com mais de 75000 entradas estava excessivamente lento para ser finalizado. Novamente, não houve nenhuma diferença significativa entre os tempos do Watershed-ng utilizando canais de comunicação via HDFS ou sistema de arquivo local.

Nessa aplicação mais complexa, Watershed-ng escala melhor que Spark, sendo até cinco vezes mais rápido para 500000 pontos. Isso aparenta ser pelo fato dos operadores em Spark executarem em uma maneira mais sincronizada, principalmente considerando que a implementação do kNN gera dependências amplas na linhagem de RDDs do Spark. Em Watershed-ng, o sequencia de computação é realizada de uma maneira mais assíncrona, com os filtros Compare e Merge sendo computados em paralelo.

5.5 Considerações finais

Com base nos resultados apresentados neste capítulo, podemos verificar que a plataforma do Watershed-ng oferece um modelo de programação que requer códigos simples no desenvolvimento das aplicações e apresenta bom desempenho quando comparado com o Hadoop, embora esperamos ser capazes de melhorar ainda mais a eficiencia na comunicação dos fluxos de dados. A complexidade de código do Watershed-ng se mos-

tra comparável ao Hadoop em aplicações que claramente se adequam ao modelo de programação MapReduce. Em aplicações mais complexas onde o modelo assíncrono do Watershed-ng pode ser melhor aproveitado, o Watershed-ng consegue superar a plataforma Spark em aplicações que envolvem dependências amplas.

Podemos observar também que a abstração modular de fluxos de dados permite maior reutilização de código sem apresentar indícios de inserir grande penalidade no tempo de execução, com base na comparação do desempenho do Watershed-ng com as demais plataformas.

Capítulo 6

Conclusões

Multiplas plataformas têm sido propostas para facilitar a tarefa de programação de aplicações que utilizam dados massivos. Entretanto, a maior parte dessas plataformas permite que os desenvolvedores definam apenas o processamento realizado nos dados — a comunicação é tratada como uma caixa preta definida internamente como parte da plataforma de processamento distribuído, à qual o desenvolvedor não possui acesso direto pela interface de programação. Neste trabalho, descrevemos o processo de re-engenharia do Watershed, um sistema de processamento de fluxos de dados, tornando os canais de comunicação de fluxos de dados objetos de primeira classe. Dessa maneira, programadores são capazes de desenvolver seus próprios canais de comunicação, de forma que melhor representem as necessidades de suas aplicações, bem como reutilizar componentes previamente desenvolvidos, compondo um canal de comunicação a partir de módulos pré-existentes.

O processo de re-engenharia nos permitiu realizar otimizações na implementação e desenvolver uma abordagem modular, aproveitando e integrando módulos disponíveis no ecosistema Apache Hadoop, como YARN e ZooKeeper. A nova abstração desenvolvida para os canais de comunicação possibilitou estender facilmente o Watershed para incluir uma interface simples para o HDFS, o sistema de arquivos distribuído do Hadoop, simplificando a tarefa do programador de obter acesso aos grandes volumes de dados a serem processados.

Os nossos resultados mostram uma redução significativa na complexidade do código, com base no número de linhas de código, assim como uma melhoria de desempenho em comparação aos dois sistemas que o precederam. Quando comparado com o Hadoop em um caso bem adaptado ao modelo de programação MapReduce, o desempenho do Watershed-ng se mostrou comparável. Em aplicações mais complexas onde o modelo assíncrono do Watershed-ng pode ser melhor aproveitado, o Watershed-ng

consegue superar a plataforma Spark em aplicações que envolvem dependências amplas.

Como trabalhos futuros pretendemos continuar o desenvolvimento de novos padrões de fluxos de dados e de novos *encoders*/*decoders*, como um agregador de mensagem para reduzir a penalidade do envio de pequenas mensagens na rede. Acreditamos que através da implementação de protocolos de comunicação mais otimizados, o desempenho apresentado pelo Watershed-ng pode melhorar ainda mais. Além disso, pretendemos desenvolver também o mecanismo de tipagem semântica dos fluxos de dados, integrado com um serviço de registro e compartilhamento de tipos, proporcionando uma melhor abstração para especificar a conexão de comunicação entre filtros. Também estamos considerando a possibilidade de integrar a plataforma do Watershed-ng com uma solução para explorar mecanismos otimizados de gerenciamento de rede, como Orchestra e Coflow, usando os padrões semânticos de fluxos de rede presentes no Watershed.

Referências Bibliográficas

Agha, G. A. (1990). *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press.

Agha, G. A.; Mason, I. A.; Smith, S. F. & Talcott, C. L. (1997). A foundation for actor computation. *Journal of Functional Programming*, 7(01):1--72.

Akidau, T.; Balikov, A.; Bekiroğlu, K.; Chernyak, S.; Haberman, J.; Lax, R.; McVeety, S.; Mills, D.; Nordstrom, P. & Whittle, S. (2013). Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033--1044.

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. ACM.

Arianfar, S.; Nikander, P. & Ott, J. (2010). On content-centric router design and implications. Em *Proceedings of the Re-Architecting the Internet Workshop*, ReARCH '10, pp. 5:1--5:6, New York, NY, USA. ACM.

Bahmani, B.; Chakrabarti, K. & Xin, D. (2011). Fast personalized pagerank on mapreduce. Em *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pp. 973--984, New York, NY, USA. ACM.

Beynon, M. D.; Ferreira, R.; Kurç, T. M.; Sussman, A. & Saltz, J. H. (2000). Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. Em *Eighth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies in cooperation with Seventeenth IEEE Symposium on Mass Storage Systems, MSS 2000, College Park, MD, USA, March 27-30, 2000*, pp. 119--134.

Bu, Y.; Howe, B.; Balazinska, M. & Ernst, M. D. (2010). Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285--296.

Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems. Em *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 335--350. USENIX Association.

Chambers, C.; Raniwala, A.; Perry, F.; Adams, S.; Henry, R. R.; Bradshaw, R. & Weizenbaum, N. (2010). Flumejava: Easy, efficient data-parallel pipelines. Em *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pp. 363--375, New York, NY, USA. ACM.

Chowdhury, M. & Stoica, I. (2012). Coflow: a networking abstraction for cluster applications. Em *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pp. 31--36. ACM.

de Souza Ramos, T. L. A.; Oliveira, R. S.; de Carvalho, A. P.; Ferreira, R. A. C. & Meira, W. (2011). Watershed: A high performance distributed stream processing system. Em *Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on*, pp. 191--198. IEEE.

Dean, J. & Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107--113.

Ferreira, R. A.; Meira, W.; Guedes, D.; Drummond, L. M. d. A.; Coutinho, B.; Teodoro, G.; Tavares, T.; Araujo, R. & Ferreira, G. T. (2005). Anthill: A scalable run-time environment for data mining applications. Em *Computer Architecture and High Performance Computing, 2005. SBAC-PAD 2005. 17th International Symposium on*, pp. 159--166. IEEE.

Ghemawat, S.; Gobioff, H. & Leung, S.-T. (2003). The google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29--43.

Hall, M.; Kirby, R. M.; Li, F.; Meyer, M.; Pascucci, V.; Phillips, J. M.; Ricci, R.; Van der Merwe, J. & Venkatasubramanian, S. (2013). Rethinking abstractions for big data: Why, where, how, and what. *arXiv preprint arXiv:1306.3295*.

Hunt, P.; Konar, M.; Junqueira, F. P. & Reed, B. (2010). Zookeeper: wait-free coordination for internet-scale systems. Em *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pp. 11--11.

Isard, M.; Budiu, M.; Yu, Y.; Birrell, A. & Fetterly, D. (2007). Dryad: Distributed data-parallel programs from sequential building blocks. Em *Proceedings of the 2007 Eurosys Conference*, Lisbon, Portugal. Association for Computing Machinery, Inc.

Li, H.; Wang, Y.; Zhang, D.; Zhang, M. & Chang, E. Y. (2008). PFP: Parallel fp-growth for query recommendation. Em *Proceedings of the 2008 ACM Conference on Recommender Systems*, RecSys '08, pp. 107--114, New York, NY, USA. ACM.

Malewicz, G.; Austern, M. H.; Bik, A. J.; Dehnert, J. C.; Horn, I.; Leiser, N. & Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. Em *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135--146. ACM.

Murthy, A.; Vavilapalli, V. K.; Eadline, D.; Markham, J. & Niemiec, J. (2013). *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*. Pearson Education.

Nagle, J. B. (1988). Innovations in internetworking. capítulo On Packet Switches with Infinite Storage, pp. 136--139. Artech House, Inc., Norwood, MA, USA.

Neumeyer, L.; Robbins, B.; Nair, A. & Kesari, A. (2010). S4: Distributed stream computing platform. Em *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pp. 170--177. IEEE.

Page, L.; Brin, S.; Motwani, R. & Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab. Previous number = SIDL-WP-1999-0120.

Qian, Z.; He, Y.; Su, C.; Wu, Z.; Zhu, H.; Zhang, T.; Zhou, L.; Yu, Y. & Zhang, Z. (2013). Timestream: Reliable stream computation in the cloud. Em *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pp. 1--14, New York, NY, USA. ACM.

Reed, B. & Junqueira, F. P. (2008). A simple totally ordered broadcast protocol. Em *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, p. 2. ACM.

Shreedhar, M. & Varghese, G. (1995). Efficient fair queueing using deficit round robin. Em *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '95, pp. 231--242, New York, NY, USA. ACM.

Shvachko, K.; Kuang, H.; Radia, S. & Chansler, R. (2010). The hadoop distributed file system. Em *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1--10. IEEE.

Toshniwal, A.; Taneja, S.; Shukla, A.; Ramasamy, K.; Patel, J. M.; Kulkarni, S.; Jackson, J.; Gade, K.; Fu, M.; Donham, J. et al. (2014). Storm@ twitter. Em *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 147--156. ACM.

Vavilapalli, V. K.; Murthy, A. C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S. et al. (2013). Apache hadoop YARN: Yet another resource negotiator. Em *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5. ACM.

Veloso, A.; Meira Jr, W.; Ferreira, R.; Neto, D. G. & Parthasarathy, S. (2004). Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining. Em *Knowledge Discovery in Databases: PKDD 2004*, pp. 422--433. Springer.

White, T. (2009). *Hadoop: the definitive guide: the definitive guide*. "O'Reilly Media, Inc.".

Wu, G.; Li, H.; Hu, X.; Bi, Y.; Zhang, J. & Wu, X. (2009). Mrec4.5: C4.5 ensemble classification with mapreduce. Em *ChinaGrid Annual Conference, 2009. ChinaGrid '09. Fourth*, pp. 249–255.

Zaharia, M.; Chowdhury, M.; Franklin, M. J.; Shenker, S. & Stoica, I. (2010). Spark: cluster computing with working sets. Em *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pp. 10--10.

Zaki, M. J. & Wagner Meira, J. (2014). *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press.

hws.core

Class Filter

java.lang.Object
    hws.core.DefaultExecutor
        hws.core.Filter

All Implemented Interfaces:

ChannelReceiver

```
public abstract class Filter
extends DefaultExecutor
implements ChannelReceiver
```

A classe Filter define a abstração de um filtro no modelo de programação do Watershed-ng. Essa classe deve ser estendida pelo usuário, implementando os métodos abstrados.

Method Summary

Methods

Modifier and Type	Method and Description
java.lang.String	**attribute**(java.lang.String attr) Obtém o valor associado com um determinado nome de atributo.
java.util.Set<java.lang.String>	**inputChannels**() Obtém o conjunto de nomes dos fluxos de entrada de dados.
int	**instanceId**() Obtém o identificado dessa instância de filtro.
java.lang.String	**name**() Obtém o nome do filtro como definido pelo usuário.
void	**onChannelHalt**(java.lang.String channelName) Esse método é disparado uma vez que todos os produtores de um determinado fluxo de entrada terminam suas execuções.
void	**onChannelsHalt**() Esse método é disparado quando todos os produtores de todos os fluxo de entrada desse filtro específico encerraram suas execuções.
ChannelOutputSet	**outputChannel**(java.lang.String channelName) Obtém um determinado fluxo de saída de dados.
java.util.Set<java.lang.String>	**outputChannels**() Obtém o conjunto de nomes dos fluxos de saída de dados.
abstract void	**process**(java.lang.String src, java.lang.Object data) O método process é um método abstrato que tem como objetivo definir a computação que será realizada para cada dado originado dos fluxos de entrada.
void	**receive**(java.lang.String src, java.lang.Object data)

Method Detail

process

```
public abstract void process(java.lang.String src,
           java.lang.Object data)
```

O método process é um método abstrato que tem como objetivo definir a computação que será realizada para cada dado originado dos fluxos de entrada. Esse método pode ser disparado de maneira concorrente, sendo necessário implementá-lo de maneira thread-safe.

Parameters:

`src` - especifica o nome do fluxo de entrada responsável por gerar o evento para o processamento do dado.

`data` - objeto do dado recebido pelo fluxo de entrada que deve ser processado.

onChannelHalt

```
public void onChannelHalt(java.lang.String channelName)
```

Esse método é disparado uma vez que todos os produtores de um determinado fluxo de entrada terminam suas execuções.

Parameters:

`channelName` - nome do fluxo de entrada encerrado.

onChannelsHalt

```
public void onChannelsHalt()
```

Esse método é disparado quando todos os produtores de todos os fluxo de entrada desse filtro específico encerraram suas execuções.

receive

```
public void receive(java.lang.String src,
           java.lang.Object data)
```

Specified by:

`receive` in interface `ChannelReceiver`

name

```
public java.lang.String name()
```

Obtém o nome do filtro como definido pelo usuário.

Returns:

Retorna o nome do filtro relacionado à essa instância de filtro.

instanceId

```
public int instanceId()
```

Obtém o identificado dessa instância de filtro.

Returns:

Retorna o inteiro identificado dessa instância.

attribute

```
public java.lang.String attribute(java.lang.String attr)
```

Obtém o valor associado com um determinado nome de atributo.

Parameters:

`attr` - nome do atributo desejado.

Returns:

Retorna o valor associado ao atributo requisitado.

inputChannels

```
public java.util.Set<java.lang.String> inputChannels()
```

Obtém o conjunto de nomes dos fluxos de entrada de dados.

Returns:

Retorna um conjunto de strings contendo o nome de cada fluxo de entrada.

outputChannels

```
public java.util.Set<java.lang.String> outputChannels()
```

Obtém o conjunto de nomes dos fluxos de saída de dados.

Returns:

Retorna um conjunto de strings contendo o nome de cada fluxo de saída.

outputChannel

```
public ChannelOutputSet outputChannel(java.lang.String channelName)
```

Obtém um determinado fluxo de saída de dados. Esse método retorna uma interface que oferece uma abstração para os fluxos de saída de dados. Os dados produzidos nessa interface são encaminhados para todos os fluxos de dados conectados à esse canal de comunicação específico, isto é, todos os fluxos conectados à porta

especificada como parâmetro.

Parameters:

`channelName` - nome do fluxo de dados para o qual se deseja produzir uma saída.

Returns:

Retorna a interface do fluxo de saída requisitado.

hws.core

Class ChannelSender

java.lang.Object
    hws.core.DefaultExecutor
        hws.core.ChannelSender

Direct Known Subclasses:

ChannelEncoder

```
public abstract class ChannelSender
extends DefaultExecutor
```

A classe ChannelSender define a abstração de um sender no modelo de programação do Watershed-ng. Essa classe deve ser estendida pelo usuário, definindo o comportamento do padrão de comunicação específico para a aplicação.

Method Summary

Methods

Modifier and Type	Method and Description
java.lang.String	**attribute**(java.lang.String key) Obtém o valor associado com um determinado nome de atributo.
java.lang.String	**channelName**() Obtém o nome do canal de comunicação como definido pelo usuário.
java.lang.String	**consumerName**() Obtém o nome do filtro consumidor como definido pelo usuário.
int	**instanceId**() Obtém o identificado dessa instância de filtro.
int	**numConsumerInstances**() Obtém o número total de instâncias do filtro consumidor conectado diretamente a esse fluxo de dados.
java.lang.String	**producerName**() Obtém o nome do filtro produtor como definido pelo usuário.
abstract void	**send**(java.lang.Object data) Esse método é chamado como parte de um fluxo de saída.

Method Detail

send

```
public abstract void send(java.lang.Object data)
```

Esse método é chamado como parte de um fluxo de saída. Esse método é responsável por definir o padrão de comunicação referente à escrita de dados no canal de comunicação.

Parameters:

`data` - objeto representando a mensagem que deve ser enviada pelo fluxo de dados.

instanceId

```
public int instanceId()
```

Obtém o identificado dessa instância de filtro.

Returns:

Retorna o inteiro identificado dessa instância.

channelName

```
public java.lang.String channelName()
```

Obtém o nome do canal de comunicação como definido pelo usuário.

Returns:

Retorna o nome desse canal de comunicação.

producerName

```
public java.lang.String producerName()
```

Obtém o nome do filtro produtor como definido pelo usuário.

Returns:

Retorna o nome do filtro produtor conectado diretamente a esse fluxo de dados.

consumerName

```
public java.lang.String consumerName()
```

Obtém o nome do filtro consumidor como definido pelo usuário.

Returns:

Retorna o nome do filtro consumidor conectado diretamente a esse fluxo de dados.

numConsumerInstances

```
public int numConsumerInstances()
```

Obtém o número total de instâncias do filtro consumidor conectado diretamente a esse fluxo de dados.

Returns:

Retorna o número total de instâncias do filtro consumidor.

attribute

```
public java.lang.String attribute(java.lang.String key)
```

Obtém o valor associado com um determinado nome de atributo.

Parameters:

`attr` - nome do atributo desejado.

Returns:

Retorna o valor associado ao atributo requisitado.

hws.core

Class ChannelDeliver

java.lang.Object
    hws.core.DefaultExecutor
        hws.core.ChannelDeliverer

Direct Known Subclasses:

ChannelDecoder

```
public abstract class ChannelDeliverer
extends DefaultExecutor
```

A classe ChannelDeliverer define a abstração de um deliverer no modelo de programação do Watershed-ng. Essa classe deve ser estendida pelo usuário, definindo o comportamento do padrão de comunicação específico para a aplicação.

Method Summary

Methods

Modifier and Type	Method and Description
java.lang.String	**attribute**(java.lang.String key) Obtém o valor associado com um determinado nome de atributo.
java.lang.String	**channelName**() Obtém o nome do canal de comunicação como definido pelo usuário.
ChannelReceiver	**channelReceiver**() Obtém o próximo elemento conectado ao fluxo de dados.
void	**deliver**(java.lang.Object data) Esse método é responsável por entregar o dado especificado por parâmetro para o próximo componente do fluxo de dados.
int	**instanceId**() Obtém o identificado dessa instância de filtro.
int	**numFilterInstances**() Obtém o número total de instâncias do filtro conectado como consumidor de dados desse fluxo.
void	**onProducersHalt**() Esse método é disparado quando todos os produtores específicos desse fluxo de entrada encerraram suas execuções.

Method Detail

channelReceiver

```
public ChannelReceiver channelReceiver()
```

Obtém o próximo elemento conectado ao fluxo de dados.

Returns:

Retorna o objeto do próximo elemento do fluxo de dados.

instanceId

```
public int instanceId()
```

Obtém o identificado dessa instância de filtro.

Returns:

Retorna o inteiro identificado dessa instância.

numFilterInstances

```
public int numFilterInstances()
```

Obtém o número total de instâncias do filtro conectado como consumidor de dados desse fluxo.

Returns:

Retorna o número total de instâncias do filtro consumidor.

channelName

```
public java.lang.String channelName()
```

Obtém o nome do canal de comunicação como definido pelo usuário.

Returns:

Retorna o nome desse canal de comunicação.

deliver

```
public void deliver(java.lang.Object data)
```

Esse método é responsável por entregar o dado especificado por parâmetro para o próximo componente do fluxo de dados. Essa interface abstrai o tipo do próximo componente, podendo ser qualquer classe que implemente um ChannelReceiver.

Parameters:

`data` - dado a ser entregado ao próximo elemento do canal de comunicação.

attribute

```
public java.lang.String attribute(java.lang.String key)
```

Obtém o valor associado com um determinado nome de atributo.

Parameters:

`attr` - nome do atributo desejado.

Returns:

Retorna o valor associado ao atributo requisitado.

onProducersHalt

```
public void onProducersHalt()
```

Esse método é disparado quando todos os produtores específicos desse fluxo de entrada encerraram suas execuções.

hws.core

Class ChannelEncoder

java.lang.Object
- hws.core.DefaultExecutor
  - hws.core.ChannelSender
    - hws.core.ChannelEncoder

```
public abstract class ChannelEncoder
extends ChannelSender
```

A classe ChannelEncoder define a abstração de um encoder no modelo de programação do Watershed-ng. Essa classe deve ser estendida pelo usuário, definindo o comportamento do padrão de comunicação específico para a aplicação.

Method Summary

Methods

Modifier and Type	Method and Description
`ChannelSender`	**`channelSender`**`()` Obtém o próximo elemento do fluxo de saída de dados.
`abstract void`	**`encode`**`(java.lang.Object data)` Esse método é chamado como parte de um fluxo de saída.
`void`	**`send`**`(java.lang.Object data)` Esse método é chamado como parte de um fluxo de saída.

Method Detail

channelSender

```
public ChannelSender channelSender()
```

Obtém o próximo elemento do fluxo de saída de dados.

Returns:

Retorna a interface do fluxo de saída de dados.

send

```
public void send(java.lang.Object data)
```

Description copied from class: `ChannelSender`
Esse método é chamado como parte de um fluxo de saída. Esse método é responsável por definir o padrão de comunicação referente à escrita de dados no canal de comunicação.

Specified by:

`send` in class `ChannelSender`

Parameters:

`data` - objeto representando a mensagem que deve ser enviada pelo fluxo de dados.

encode

```
public abstract void encode(java.lang.Object data)
```

Esse método é chamado como parte de um fluxo de saída. Esse método é responsável por realizar alguma transformação, geralmente relacionada a codificação do dado, e está relacionado à escrita de dados no canal de comunicação.

Parameters:

`data` - objeto representando a mensagem que deve ser explicitamente propagada pelo fluxo de dados, após receber alguma transformação.

hws.core

Class ChannelDecoder

java.lang.Object
    hws.core.DefaultExecutor
        hws.core.ChannelDeliverer
            hws.core.ChannelDecoder

All Implemented Interfaces:

ChannelReceiver

```
public abstract class ChannelDecoder
extends ChannelDeliverer
implements ChannelReceiver
```

A classe ChannelDecoder define a abstração de um decoder no modelo de programação do Watershed-ng. Essa classe deve ser estendida pelo usuário, definindo o comportamento do padrão de comunicação específico para a aplicação.

Method Summary

Methods

Modifier and Type	Method and Description
`abstract void`	**`decode`**`(java.lang.Object data)` Esse método é chamado como parte de um fluxo de saída.

Method Detail

decode

```
public abstract void decode(java.lang.Object data)
```

Esse método é chamado como parte de um fluxo de saída. Esse método é responsável por realizar alguma transformação, geralmente relacionada a decodificação do dado, e está relacionado à leitura de dados no canal de comunicação.

Parameters:

`data` - objeto representando a mensagem que deve ser explicitamente propagada pelo fluxo de dados, após receber alguma transformação.