UNIVERSIDADE CATÓLICA DE PERNAMBUCO
CENTRO DE CIÊNCIAS E TECNOLOGIA
TRABALHO DE CONCLUSÃO DE CURSO

PROCESSAMENTO DE DADOS EM LARGA ESCALA NA COMPUTAÇÃO DISTRIBUÍDA

por

CELSO LUIZ AGRA DE SÁ FILHO

Recife, Junho de 2011

UNIVERSIDADE CATÓLICA DE PERNAMBUCO
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO

Processamento de dados em larga escala na computação distribuída

por

Celso Luiz Agra de Sá Filho

Monografia apresentada ao curso de Ciência da computação da Universidade Católica de Pernambuco, como parte dos requisitos necessários à obtenção do grau de Bacharel em Ciência da Computação.

ORIENTADOR: Silvio Soares Bandeira, Mestre

Recife, Junho de 2011

"Sempre faço o que não consigo fazer para aprender o que não sei"

Pablo Picasso

Resumo da Monografia apresentada ao curso de Ciência da Computação da Universidade Católica de Pernambuco.

PROCESSAMENTO DE DADOS EM LARGA ESCALA NA COMPUTAÇÃO DISTRIBUÍDA

Celso Luiz Agra de Sá Filho

Junho/2011

Orientador: Silvio Soares Bandeira, Mestre.
Área de Concentração: Computação Distribuída.
Palavras-chave: **MapReduce, GFS, Hadoop.**
Número de Páginas: 42.

Este documento visa apresentar soluções para o problema do processamento de grandes conjuntos de dados, com ênfase no paradigma *MapReduce*. Um Modelo abstrato de programação que auxilia na execução de operações em ambientes distribuídos. Focando na alta capacidade de manipulação de informações em larga escala, utilizando uma organização definida por clusters. O uso desse paradigma está se popularizando de forma gradativa devido aos benefícios que ela traz, tanto no desenvolvimento de aplicações quanto na *performance* das operações nela contida. Agora, um sistema pode ser desenvolvido para que o programador não necessite ter conhecimento sobre ambientes distribuídos. Sendo assim, a facilidade de criar aplicações distribuídas contribuiu para o sucesso desse modelo. Atualmente, aplicações como o Google File system (GFS) e o Hadoop, ganharam bastante notoriedade pela sua eficiência e desempenho. Dessa forma, foram adotadas por grandes corporações como a Google (precursor do *MapReduce*), Facebook, Twitter, entre outras. O principal interesse dessas empresas está na alta capacidade de alocação de dados e no seu processamento.

Abstract of Dissertation presented to UNICAP.

LARGE DATA PROCESSING ON DISTRIBUTED COMPUTING

Celso Luiz Agra de Sá Filho

June/2011

Supervisor(s): Silvio Soares Bandeira, Master
Area of Concentration: Distributed Computing.
Keywords: **MapReduce, GFS, Hadoop.**
Number of Pages: 42.

This document presents solutions to the problem of processing large data, with emphasis on MapReduce paradigm. An abstract programming model that helps in the implementation of methods on distributed environments. Focusing on high capacity for handling information with large scale, using an environment defined by clusters. This paradigm is becoming popular because it has many benefits, both in the development of applications and in performance. Now, a system can be implemented in a way that the developer does not need to have knowledge about the environment. Thus, the facility of creating distributed applications, contributed to the success of this model. Applications like the Google File System (GFS) and Hadoop, gained enough notoriety for its efficiency and performance. This model was adopted by large corporations like Google (forerunner of MapReduce), Facebook, Twitter, among others. The main interest of these companies is the high capacity of data allocation and processing.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

A era da informação, surgiu com o intuito de contribuir com a sociedade em sua busca por conhecimento. Essa necessidade só foi possível ser suprida, devido a evolução dos novos dispositivos eletrônicos, que puderam fornecer meios para amadurecer cada vez mais as idéias de um determinado conjunto de pessoas. Nos dias atuais, torna-se inevitável o surgimento gradativo de novos dados no universo da tecnologia da informação (TI). Segundo White (2009, p. 1): "Nós vivemos na era dos dados". A partir dessa afirmação, é notável a preocupação do momento atual, pois é necessário armazenar um alto volume de informações. Sendo assim, as máquinas precisam fornecer melhores condições para que esses dados possam ser manipulados de uma maneira simples e eficaz, aproveitando ao máximo toda a capacidade que um ambiente computacional pode oferecer.

No mundo real, sistemas são utilizados diariamente por inúmeros usuários, que lêem e escrevem constantemente dados, gerando uma alta taxa de transferência (*throughput*). Portanto, é indispensável o uso de mecanismos capazes de suportar diversas solicitações, como no caso dos sistemas distribuídos, que utilizam o conceito da redundância de componentes, aumentando a escalabilidade, a segurança e outros benefícios que serão abordados posteriormente. Atualmente, esses modelos estão sendo adotados por grandes corporações, como um meio para fornecer um melhor desempenho às suas aplicações. Sistemas como o Facebook chegaram a armazenar 1 *petabyte* de informações no ano de 2008, registrando cerca de10 bilhões de fotos em seus servidores (WHITE, 2009, p. 1). Outro exemplo encontra-se no *site* da Organização Européia de Pesquisa Nuclear (CERN), onde afirmam que o acelerador de partículas LHC (Grande Colisor de Hádrons) poderá produzir dados de até 15 *petabytes* ao ano[1]. De acordo com os exemplos citados, pode-se concluir a real necessidade do uso de sistemas robustos com um alto poder de armazenamento e de processamento dos dados nele contido.

É notável que ao longo do tempo, os computadores evoluíram seu poder de armazenamento de *bytes* para *terabytes*. Esse desenvolvimento trouxe grandes benefícios aos usuários, que puderam desfrutar de uma elevada quantidade de espaço para armazenar seus dados. Contudo, o grande problema em questão não se baseia no tamanho das informações armazenadas, e sim na quantidade de dados que podem ser processados pelo sistema. Tendo

[1] http://public.web.cern.ch/public/en/LHC/Computing-en.html

ainda como exemplo a rede social Facebook, estima-se que milhões de usuários acessem diariamente o *site*, lendo e escrevendo inúmeras informações em frações de segundo. Para esse problema, é preciso utilizar modelos eficazes (abordados em seções posteriores), capazes de classificar os dados para que o sistema possa manipulá-los de forma rápida e eficiente.

A principal motivação pelo qual foi escolhido este tema é a preocupação da era atual, relacionado ao crescimento desenfreado da quantidade de dados armazenados em todo o universo da TI. Mecanismos como a mineração de dados (*Data Mining*) estão sendo bastante utilizados atualmente, acumulando grandes quantidades de informações. Muitas empresas utilizam técnicas baseadas em *logs* de atividades, que especificam as diversas ações realizadas por um único usuário. Segundo Lin (2010, p. 1), essa prática pode trazer sérias conseqüências quando utilizada de forma errada. Uma empresa pode adquirir tantas informações em pouco tempo, que não conseguirá processá-las e nem organizá-las. Dessa forma, acabarão desprezando-as, perdendo oportunidades que poderiam auxiliar na evolução do seu sistema.

Portanto, torna-se necessário que as máquinas possuam um *hardware* com alto poder de processamento, para realizar as inúmeras atividades solicitadas pelo usuário. Gordon Moore foi um dos primeiros a expressar tal preocupação, em seu artigo "*Cramming more components onto integrated circuit*[2]". Ele afirmou que desde a década de 50, a prática de miniaturizar componentes eletrônicos, para incluir funcionalidades mais complexas em menos espaço, vem sendo bastante difundida entre todos os pesquisadores da sua época. Dessa forma, a cada 18 meses[3] o nível de transistores colocados em um circuito integrado iria dobrar, ou seja, o nível complexidade aumentaria nos dispositivos. Esse documento ficou conhecido como a Lei de Moore[4]. Por muito tempo, essa teoria esteve certa. Sistemas necessitavam de supercomputadores com inúmeros componentes que pudessem realizar uma vasta quantidade de cálculos em pouco tempo, fornecendo uma estrutura para que o sistema pudesse operar com um melhor desempenho.

Recentemente este paradigma foi quebrado devido ao avanço dos componentes e a criação de ambientes paralelos e distribuídos, que puderam fornecer um alto desempenho se comparado aos sistemas centralizados. Agora, uma aplicação poderia ser executada concorrentemente entre os diversos elementos de uma maquina, seja ela distribuída ou multiprocessada. As técnicas de processamento de grandes quantidades de dados serão abordados no próximo capitulo (Seção 2), que deverão apresentar conceitos e estratégias

[2] Em português: Agrupando mais componentes em circuitos integrados.
[3] Moore atualizou essa estimativa para 2 anos.
[4] ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf

utilizados para auxiliar na manipulação desses dados. Desde a origem da técnica de divisão e conquista até as aplicações da computação nas nuvens.

No capitulo posterior (Seção 3) deverá abordar técnicas do novo paradigma da computação distribuída, o *MapReduce*. Um sistema capaz de processar uma grande quantidade de dados em apenas poucos segundos, utilizando uma estrutura hierárquica e escalar, de fácil usabilidade. Após a apresentação desse novo paradigma, serão abordados alguns estudos de caso (Seção 4), que obtiveram sucesso com a utilização desse modelo, dando ênfase as duas aplicações mais populares que adotaram esse mecanismo, o Hadoop e o Google File System.

2 TÉCNICAS DE PROCESSAMENTO DE DADOS

A era da informação proporcionou grandes benefícios, fornecendo máquinas capazes de realizar com alta velocidade, milhares de cálculos. Na medida em que evoluíam, os dispositivos ficavam cada vez mais robustos, suportando cargas elevadas de informações. A melhoria dos recursos de um componente, tal como a evolução gradativa da capacidade de armazenamento e processamento de uma simples máquina, aumenta o nível de complexidade dos dispositivos, como previsto por Moore, ou seja, um simples computador deverá estar em constante evolução para atender as necessidades de um sistema.

As técnicas de processamento de dados apresentadas nesta seção, visam proporcionar maneiras simples e eficientes de solucionar este problema. É possível criar um sistema robusto, que possa atender as demandas impostas pelos usuários. A primeira idéia a se pensar é como podemos implantar sistemas sem a necessidade de utilizar máquinas de grande porte, tais como um *mainframe* ou um supercomputador. Essa questão será abordada durante toda a seção, apresentando soluções quanto à forma de processamento dessas informações. Vale ressaltar que a grande dificuldade destacada por esse documento é a necessidade de processar dados em larga escala e por isso é necessária a utilização de sistemas de alto desempenho.

Uma maneira de melhorar a *performance* de uma aplicação é a separação dos problemas em partes independentes. A divisão dessas atividades contribui com a resolução das dificuldades impostas pelo sistema, auxiliando em questões como a execução paralela dos processos e o controle do nível de complexidade de uma aplicação.

Nesta seção serão abordados mecanismos para a realização do processamento de dados de forma rápida e eficiente. Utilizando técnicas para a divisão de atividades, conceitos de paralelismo e mecanismos de distribuição de problemas entre os componentes envolvidos.

2.1 Dividir para Conquistar

As técnicas de "divisão e conquista" foram utilizadas primeiramente por Karatsuba Anatolii[5], que desenvolveu um algoritmo, apresentado em um documento intitulado "The

[5] http://www.mi.ras.ru/~karatsuba/index_e.html

complexity of Computations[6]", que visa solucionar problemas matemáticos a partir da separação dos cálculos, realizando atividades com um menor grau de dificuldade. A separação de problemas em partes menores foi utilizada em ambientes computacionais com o intuito de controlar a complexidade de um determinado algoritmo. É possível separar as ações em etapas, para que possam ser utilizadas em fins de legibilidade do código, desempenho e modularidade do sistema. Um sistema modular é composto por vários procedimentos que visam alcançar um determinado objetivo. A principal vantagem de desse modelo simplista, é a capacidade de reutilizar as atividades impostas dentro de um processo.

A utilização do conceito "dividir para conquistar" também pode ser aplicada para solucionar problemas do mundo real. Como no caso de um processo de desenvolvimento de software, que deverá seguir uma sequência de etapas até que o código esteja condizente com as necessidades do cliente. Esses passos são executados de forma distinta a fim de estabelecer um controle sobre o fluxo de atividades para a implementação de uma nova aplicação. Outro exemplo pode ser visto em táticas militares, adotadas com o intuito de alcançar seus objetivos. Um exército poderia utilizar estratégias para separar seus inimigos e derrotá-los com menor esforço. Assim como nos campos de batalha, a computação utiliza-se desta técnica para resolver problemas de forma eficiente e eficaz. Na medida em que um problema é dividido, sua complexidade torna-se controlável, resultando em uma solução clara e concisa da atividade.

As técnicas de divisão e conquista são comumente utilizadas em diversos algoritmos e mostraram-se a maneira mais eficiente de executar atividades que necessitem de uma vasta quantidade de execuções de comandos. Em destaque pode-se verificar o uso desse mecanismo em algoritmos, como no caso do *merge sort* (CORMEN et al., 2002, p. 28), que visa ordenar os valores separando-os em pequenos vetores e combinando-os na medida em que são ordenados. O grande recurso por trás dos algoritmos que utilizam esse princípio é a prática da recursividade, garantindo que as funções possam realizar chamadas a si mesmo, caracterizando um processo repetitivo.

A utilização da técnica de divisão e conquista não está restrito apenas a algoritmos, funções matemáticas ou estratégias militares. A utilização deste recurso também influenciou a criação de sistemas capazes de separar e executar tarefas, como os sistemas paralelos e distribuídos.

[6] http://www.ccas.ru/personal/karatsuba/divcen.pdf

2.2 Computação Paralela

A computação paralela surgiu com o intuito de atender à necessidade de aumentar o desempenho dos sistemas. Atualmente a capacidade de um processador chegou ao seu limite (ao limite da tecnologia atual). A quantidade de componentes encontrados em um processador é bastante alta, a única maneira encontrada de melhorar o desempenho de uma maquina é inserindo novos recursos, ou seja, um novo processador.

A utilização de múltiplos processadores em computadores pessoais está se tornando cada vez mais comum, devido à necessidade de processar uma quantidade maior de dados. Sistemas estão executando tarefas cada vez mais complexas e necessitam de maior velocidade em suas atividades. Um computador com multiprocessadores pode executar tarefas de forma concorrente, onde cada processador poderá executar uma atividade solicitada, contribuindo com o desempenho do sistema. A Figura 2.1 apresenta uma situação comum, onde um sistema opera com mais de um processador.

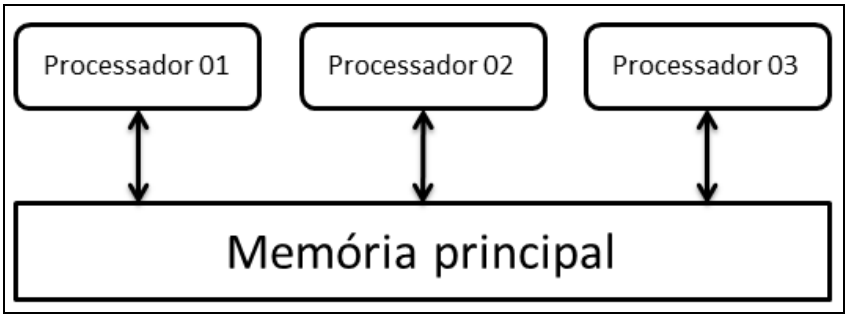

Figura 2.1: Organização de uma arquitetura de sistema paralelo.

É possível notar que este tipo de arquitetura utiliza vários processadores operando de forma concorrente e compartilhando os mesmos recursos, como no caso da memória principal. A vantagem da utilização de um sistema fortemente acoplado (como também pode ser chamado), é a garantia de um ambiente com maior desempenho. A quantidade de processadores está proporcionalmente ligada ao *throughput*. Outra vantagem da utilização de sistemas paralelos está na segurança das operações. No caso de uma falha em um dos processadores, os demais automaticamente continuarão em funcionamento. Dessa forma, o sistema continuará a execução de suas atividades, porem com um menor poder de processamento.

Um sistema fortemente acoplado pode ser classificado de acordo com a divisão de funções dentro de um ambiente. No caso dos sistemas de multiprocessamento paralelo

(SMP[7]), os processadores possuem um único nível em comum. Para os sistemas de multiprocessamento assimétrico (AMP[8]), os processadores são classificados de forma hierárquica, através do relacionamento mestre-escravo (SILBERSCHATZ et al. 2002, p. 8-9). Todo o ambiente é controlado por um único processador que envia solicitações para os demais. As diferenças entre os modelos simétrico e assimétrico podem ser visualizadas, quando comparadas em questões de *hardware* e *software*.

2.3 Computação Distribuída

A computação distribuída consiste na utilização de um conjunto de máquinas conectadas por uma rede de comunicação, atuando como um único sistema. Esse conjunto de dispositivos, ao contrário dos sistemas paralelos, possuem seus próprios recursos, como a memoria principal e o *clock* do processador. Sendo a comunicação estabelecida por uma rede através de protocolos específicos. Um sistema distribuído visa atender às características relevantes desse modelo, ou seja, precisa ser transparente, tolerante a falhas, escalar e ter um alto poder de processamento[9].

A principal motivação para a criação de uma aplicação distribuída é a sua capacidade de fornecer a um grupo de usuários, os vários recursos mantidos por ela, garantindo uma melhor disponibilidade e confiabilidade das informações e dos serviços contidos no sistema. Outra vantagem dos sistemas distribuídos é o baixo custo se comparado a um modelo que utiliza máquinas *high-end*[10], como no caso de um supercomputador. É possível conectar um conjunto de maquinas de baixo custo (denominadas *commodities*), para que possam compor um sistema robusto e eficiente.

Um modelo de sistema distribuído pode comparar-se a vários cenários, como por exemplo, um ambiente de construção civil. Cada funcionário, assim como os computadores, desempenha um papel importante para a obra. Para o cliente, não importa a quantidade de trabalhadores ou quais ferramentas estão sendo utilizadas. O importante para este exemplo, é que o objetivo seja alcançado, neste caso, a construção precisa ser concluída.

Um sistema distribuído caracteriza-se principalmente por uma camada denominada *middleware*, que provê transparência ao sistema. A utilização dessa camada fornece a

[7] Symmetric Multiprocessing
[8] Asymmetric Multiprocessing
[9] No caso dos clusters, que serão explicados ainda nesta seção.
[10] Os computadores high-end são caracterizados por serem produtos robustos e de alta capacidade.

capacidade de uma aplicação abstrair todas as informações a respeito da estrutura de um ambiente distribuído para o usuário. Sendo assim, um conjunto heterogêneo de máquinas, pode compor um único sistema, independente das questões de localização, de *hardware* ou de *software*. Como apresentado na Figura 2.2, a camada *middleware* está situada logicamente entre a camada do sistema operacional e da aplicação. É possível que diferentes programas possam estabelecer uma comunicação entre si, comutando diversas informações através de protocolos de comunicação, garantindo que sistemas distintos possam realizar troca de dados.

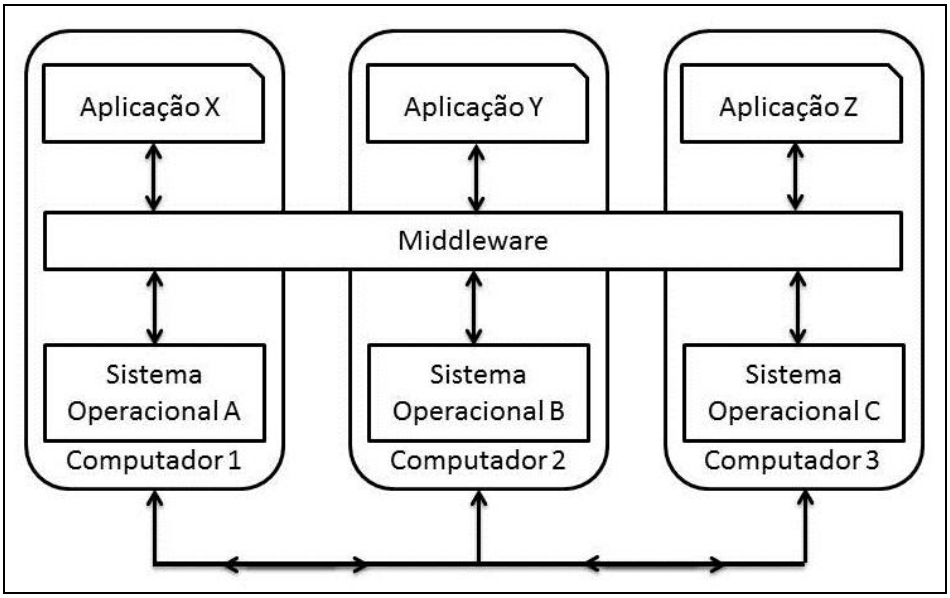

Figura 2.2: Modelo de sistema distribuído utilizando uma camada *middleware*. Essa organização garante a interoperabilidade do sistema.

2.3.1 Comunicação

Toda a estrutura de um sistema distribuído só é garantida devido ao mecanismo de comunicação entre os processos. É de extrema importância que esse modelo utilize técnicas para a comutação de informações. Os principais modelos de comunicação destacam-se pela transmissão de mensagens de baixo nível entre os diversos dispositivos. A utilização da troca de dados em uma aplicação distribuída visa realizar chamadas como se estivessem executando um serviço localmente. Assim como nos modelos centralizados, é preciso garantir que vários processos possam compartilhar informações dentro do sistema. Porém a implementação desses serviços em ambientes distribuídos são bastante complexos, e necessitam da utilização de protocolos de comunicação para que dispositivos de diferentes espécies possam comunicar-se.

O mecanismo mais comum, utilizado para a troca de informações é conhecido como chamada de procedimento remoto (RPC), que estabelece uma comunicação com o servidor de forma transparente, como se executasse uma chamada local. Primeiramente o cliente solicita uma função através de um mecanismo denominado *client stub*. A solicitação é empacotada (*marshalling*) e então enviada ao servidor, podendo passar também parâmetros dentro desse método. Depois de capturar a mensagem, o computador que possui a implementação dessa chamada deverá desempacotá-la (*unmarshalling*) para poder processá-la. A solicitação é retornada ao cliente juntamente com os dados solicitados através do *server stub* . A chamada RPC é ilustrada na Figura 2.3.

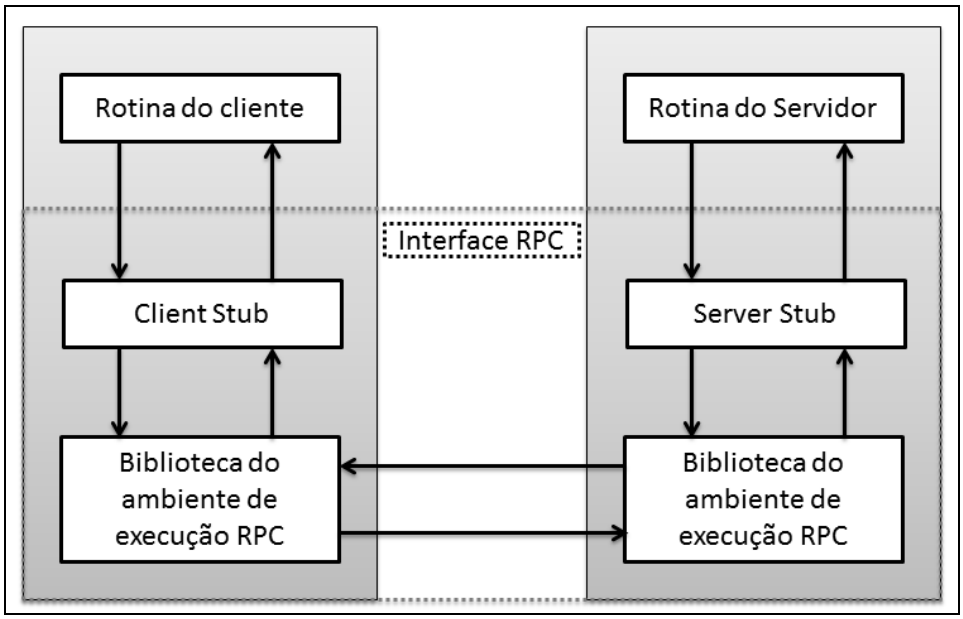

Figura 2.3: Exemplo de modelo de chamada RPC. Toda a requisição é feia por uma biblioteca do ambiente de execução RPC (RPC *Runtime Library*).

Outros mecanismos de comunicação remota também se destacam, como no caso dos *sockets*, que consistem em um mecanismo para estabelecer a troca de pacotes entre os sistemas, através da camada IP. Esse artificio é definido como uma extremidade de um canal de comunicação (SILBERSCHATZ et al. 1999, p. 355), que utiliza um *socket* para cada processo, estabelecendo uma comunicação entre eles. A comutação de mensagens é estabelecida utilizando uma arquitetura cliente-servidor, onde o servidor aguarda por uma solicitação do cliente em uma determinada porta. A mensagem transmitida é composta por um endereço IP juntamente com a porta de acesso a um serviço, podendo utilizar protocolos orientados a conexão (Ex.: TCP) ou sem conexão (Ex.: UDP) (SILBERSCHATZ et al. 1999, p. 356).

2.3.2 Classificação dos Sistemas Distribuídos

Os sistemas distribuídos podem ser classificados quanto à sua organização e a finalidade com que são utilizados. Na computação em *cluster*, um sistema é composto por um conjunto de dispositivos idênticos (tanto pelo *hardware* quanto pelo *software* utilizado), visando o alto desempenho. Portanto, é garantido também um baixo custo devido à utilização de máquinas *commodities*. A principal característica desse modelo é a sua homogeneidade entre os dispositivos. É necessário que os computadores possam se comunicar de forma rápida e eficiente, inclusive realizando transferência de processos entre os diversos dispositivos.

A disponibilidade é uma das principais vantagens desse sistema, pelo fato de garantir a tolerância a falhas. Caso uma máquina venha a falhar, o sistema continuará em execução, sem que o usuário perceba a ocorrência do problema. Porém se a falha ocorrer em determinadas regiões, como no caso de uma falta de energia, então o sistema poderá não funcionar corretamente.

No caso da computação em grade (*grid*), o ambiente é baseado na dispersão das funcionalidades de cada servidor. A principal característica desse modelo é a especialização funcional, proporcionando uma maior segurança aos serviços. Este tipo de modelo pode ser estruturado para que vários dispositivos atuem de forma conjunta, constituindo uma única aplicação. Cada máquina ficará encarregada de fornecer um determinado serviço. O principal objetivo para esse tipo de ambiente é garantir a disponibilidade e confiabilidade dos serviços. Mesmo que ocorra uma falha em determinadas regiões, o sistema poderá continuar em execução.

2.3.3 Sistema de Arquivos Distribuídos – DFS

Um sistema de arquivos fornece uma estrutura que apresenta a possibilidade de armazenar dados, disponibilizando serviços para nomeação, alteração e compartilhamento de dados, garantindo uma estrutura que determina a autenticidade de um arquivo, como no caso de um identificador, visando também a segurança quanto ao acesso desses dados.

Assim como em um ambiente distribuído, um DFS deve garantir que as particularidades dessas aplicações sejam atendidas, como no caso da transparência, da escalabilidade, da segurança, da tolerância a falhas e da heterogeneidade. Um sistema de arquivos distribuídos

deve garantir que várias maquinas estejam conectadas comutando informações através de protocolos.

Alguns mecanismos são utilizados para melhorar o desempenho deste tipo de sistema, como no caso da replicação. Além de garantir a segurança, a técnica de replicar dados pelo sistema, garante uma maior disponibilidade, realizando também o balanceamento de carga entre os dispositivos. Outra maneira para auxiliar na *performance* do sistema é o uso do esquema de *caching*. As informações são armazenadas em *cache*, para que quando solicitadas, o cliente possa recuperá-la rapidamente. Vale ressaltar que em um DFS é necessário que as informações estejam consistentes, para isso é realizado verificações sempre que houver alguma atualização nos arquivos.

Existem vários sistemas DFS. Um dos mais conhecidos é denominado o *Network File System* (NFS), que tem como finalidade fornecer acesso, armazenamento e compartilhamento de arquivos remotos, através de uma *Local Area Network* (LAN) ou *Wide Area Network* (WAN). Trata-se de um sistema que utiliza uma arquitetura cliente-servidor, onde diversas de maquinas estão conectadas por uma rede, permitindo realizar operações especificas do sistema de arquivos comum. A comunicação do NFS ocorre a partir de um conjunto de RPC's que fornece diversos serviços, como a consulta, a leitura e a escrita de informações, além de manipular *links* e diretórios controlando o acesso aos dados.

A arquitetura do NFS possui três camadas como ilustrada na Figura 2.4: a primeira camada representa a *interface* do sistema de arquivos, que deverá realizar operações comuns, tais como, *open*, *close*, *read* e *write*. A segunda camada é conhecida como *Virtual File System* (VFS), que realiza funções importantes ao sistema. Sendo assim, esta camada irá realizar operações especificas para a manipulação de dados de forma clara e concisa, sempre fornecendo transparência a este modelo. A terceira camada representa o sistema de arquivos local, que realiza a manipulação dos arquivos locais.

Além do NFS, existem outros tipos de DFS, como no caso do *Andrew File System*[11] (AFS), desenvolvido pela Universidade de Carnegie Mellon[12] (CMU), e mais tarde comercializado pela Transarc em 1989. Assim como todo sistema de arquivos distribuídos, ele proporciona vários benefícios provenientes da computação distribuída. O AFS fornece serviços de *caching*, escalabilidade, controle de acesso e simplificação na administração dos arquivos.

[11] http://www.cmu.edu/corporate/news/2007/features/andrew/what_is_andrew.shtml
[12] http://www.cmu.edu

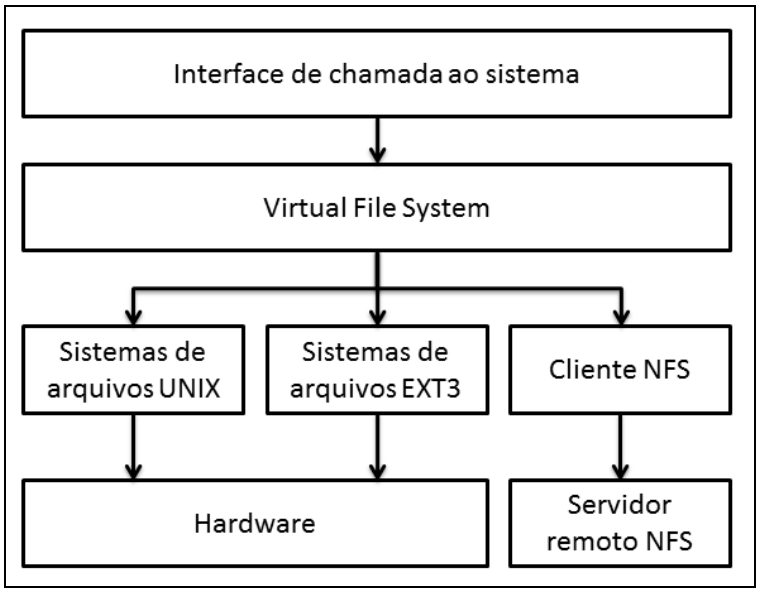

Figura 2.4: Organização do modelo NFS. As solicitações são feitas através da camada de *interface* de chamada ao sistema, que deverá chamar o VFS. Este por sua vez deverá solicitar a operação ao sistema de arquivos, podendo ser realizada localmente ou remotamente.

2.4 Computação nas Nuvens

A computação nas nuvens (*Cloud Computing*) consiste na disponibilização de serviços dentro do universo da *Internet*. Neste caso, todo e qualquer sistema pode ser abstraído como parte integrante de uma nuvem. Esse tipo de modelo é produto da evolução dos sistemas, que puderam adquirir uma alta capacidade em seu poder de processamento, melhorando o desempenho de suas atividades e a sua alta escalabilidade. A implantação de ambientes distribuídos trouxe uma maior disponibilidade aos serviços inclusos nessa nuvem. Agora, uma aplicação poderia ser solicitada por milhões de pessoas, acessando vários sistemas que se comunicam e trocam informações.

Uma nuvem pode ser classificada de três maneiras distintas: a primeira é classificada como nuvem pública (*Public Cloud*), que visa atender as solicitações públicas e gratuitas dos clientes. A segunda pode ser classificada como privada (*Private Cloud*), devido ao fato de utilizar uma infraestrutura para fornecer serviços pagos ou de uso interno de uma corporação. A junção dessas duas nuvens é conhecida como híbrida (*Hybrid Cloud*), que utiliza características das nuvens privadas e publicas.

O termo “nuvem” é proveniente da expressão metafórica utilizada para caracterizar o modelo da *web* (VELTE, A.;VELTE, T.;ELSENPETER, 2010, p. 3), onde todos os

dispositivos encontram-se interligados por uma rede de comunicação. A computação funciona de forma semelhante à *Internet*, todos os dispositivos estão interligados por uma nuvem, compartilhando informações entre diversos sistemas de diferentes gêneros. A Figura 2.5 ilustra o modelo *Cloud Computing*, onde vários elementos estão conectados por uma nuvem. Cada componente deste ambiente é caracterizado por uma funcionalidade, como os clientes e os sistemas (ou aplicações).

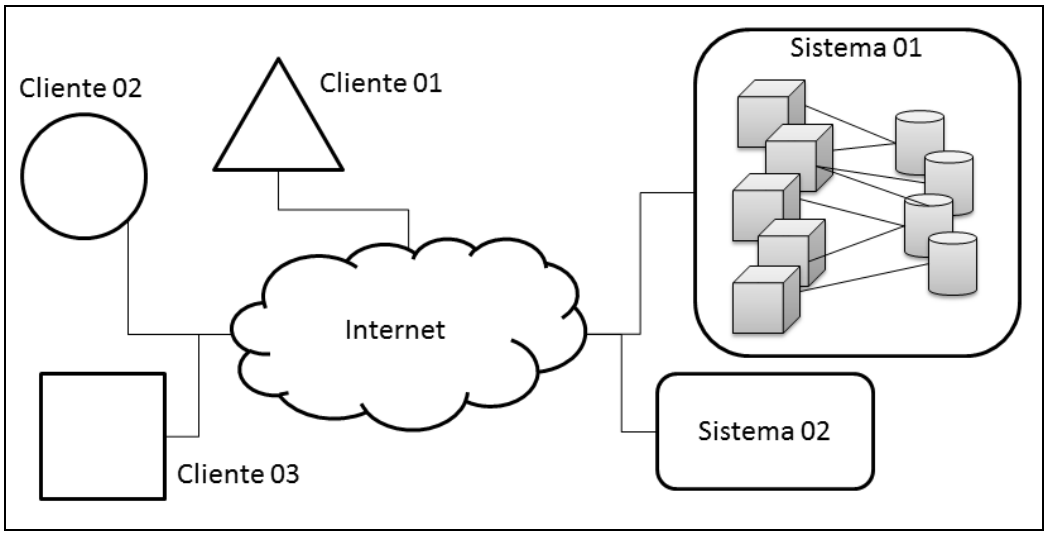

Figura 2.5: Modelo *Cloud Computing*. Cada cliente pode realizar requisições de serviços de diversos sistemas através da *internet*. Todos conectados a uma única núvem.

Um cliente consiste em um dispositivo de qualquer natureza (Ex.: um *smartphone*, um *notebook* ou um *tablet*) conectado a uma rede, que realizará solicitações aos sistemas. Este por sua vez, desempenha o papel de disponibilizar serviços para a manipulação, compartilhamento e armazenamento de dados. Grandes corporações adotaram este modelo para fornecer variados serviços. Algumas aplicações adotaram o sistema conhecido como "pague o que consumir" (*pay-as-you-go*), onde um cliente poderá usufruir de uma aplicação de uso privado. Essa visão é semelhante a qualquer serviço do mundo real, onde uma pessoa paga somente àquilo que foi utilizado, como o serviço de energia elétrica.

A nuvem pode fornecer diversos tipos de serviços, que pode ser classificados por três categorias (ANTONOPOULOS, 2010, p. 4-5; VELTE, 2010, p. 11-16):

- **Infra-estrutura como um serviço (IaaS[13]):** Fornece um ambiente para que o usuário possa desfrutar de uma estrutura fornecida na *web*, como

[13] *Infrastructure as a service*.

armazenamento ou até mesmo como um sistema operacional, através da virtualização. Um cliente pode alugar uma determinada estrutura, escolhendo a capacidade do seu *hardware*, como memória principal, espaço de armazenamento e até a capacidade do processador.

- **Plataforma como um serviço (PaaS[14]):** Oferece um ambiente capaz de auxiliar no desenvolvimento de uma aplicação, sem a utilização de um programa especifico. Atualmente diversos sistemas fornecem sua própria API[15] para que o usuário possa criar aplicações e disponibilizá-las na *web*.

- **Software como um serviço (SaaS[16]):** Um programa pode ser utilizado como um serviço disponibilizado na nuvem, onde o cliente poderá desfrutar da aplicação apenas utilizando o navegador. O exemplo mais comum desse modelo são os editores de texto, que se popularizou rapidamente pela sua facilidade de fornecer um serviço onde vários usuários podem acessar e alterar a mesma informação de forma consistente.

A principal característica desses serviços é a facilidade do uso dessas aplicações apenas utilizando o navegador como programa local. Todo o sistema está dentro de uma nuvem, podendo ser acessada em qualquer local. Essa facilidade permitiu a popularização desses sistemas, onde o cliente não se preocupa com questões de acessibilidade ou segurança. Toda sua informação está armazenada na *web*.

O uso da computação nas nuvens está se tornando cada vez mais popular entre os diversos usuários. Este tipo de aplicação fornece serviços que utilizam um alto poder de resposta apenas utilizando um simples navegador. A tendência para esse modelo, é que cada vez mais as aplicações estarão sendo incluídas nessa nuvem, que dará espaço a aplicações mais robustas e complexas, como no caso de um sistema operacional.

[14] *Plataform as a service*.
[15] *Application Programming Interface*.
[16] *Software as a service*.

3 O MODELO DE PROGRAMAÇÃO MAPREDUCE

Desde o inicio da era da informação, a busca por maior quantidade de armazenamento era imprescindível. Há quem diga que na década de 80, 640KB eram necessários para guardar todas suas informações[17]. Atualmente já estamos falando de muitos *gigabytes* para armazenar informações de um único usuário. Sendo assim é necessário utilizar meios capazes de processar uma grande quantidade de dados. Qualquer aplicação pode armazenar inúmeras informações dos seus usuários ou diversos arquivos do sistema. Para aumentar a escalabilidade e o seu poder de processamento, é necessário criar uma estrutura capaz de agrupar diversos dispositivos, podendo ser um conjunto de processadores para sistemas fortemente acoplados ou um conjunto de dispositivos no caso dos sistemas fracamente acoplados.

O uso de ambientes distribuídos está se tornando essencial para atender às necessidades dos usuários. Uma simples aplicação de grande porte necessita de centenas ou até milhares de servidores para suportar todas as solicitações dos clientes. A criação de sistemas dessa natureza é algo bastante complexo, pois é extremamente importante atender a todos os requisitos necessários de um ambiente distribuído, tal como a transparência, a segurança, a escalabilidade e a concorrência de componentes. Porém o desenvolvimento dessas aplicações é bastante complexo, pois demanda um alto conhecimento do funcionamento desses ambientes. É nessa visão que iremos abordar novas técnicas, dando ênfase ao *MapReduce*. Um modelo que está se tornando popular entre as grandes corporações, utilizando estratégias de desenvolvimento de aplicações fornecendo uma estrutura capaz de manipular, gerenciar e processar dados em sistemas distribuídos.

3.1 Motivação

Quando um programa é desenvolvido em um ambiente distribuído ou multiprocessado, o programador precisa estar ciente de toda estrutura do sistema, especificando as referências de cada dispositivo, para que possa estabelecer uma comunicação e sincronização entre os

[17] Palavras proferidas por Bill Gates em um discurso sobre softwares e a indústria da computação, onde ele afirma que o aumento da memoria de 64Kb para 640 Kb seria um grande salto, porém disse sentir-se arrependido pois em pouco tempo, usuários reclamavam da quantidade mínima de espaço encontrada no sistema. Disponivel em <http://www.csclub.uwaterloo.ca/media/1989 Bill Gates Talk on Microsoft.html>.

processos. A tarefa do desenvolvedor de criar um sistema transparente é bastante complexa, para isso são utilizadas técnicas para "esconder" toda a estrutura dessas aplicações, tal como o OpenMP (*Open Multi-Processing*) e o MPI (*Message Passing Interface*), que são utilizados com a finalidade de auxiliar o programador a fornecer transparência a uma aplicação, seja ela paralela ou distribuída.

A OpenMP é uma *interface* para a programação de aplicativos (API) que utiliza a linguagem C/C++, cuja finalidade é fornecer uma estrutura para auxiliar no desenvolvimento de aplicações escaláveis em sistemas paralelos. Agindo como se o programador utilizasse o mesmo mecanismo da memória compartilhada. No caso da MPI, trata-se de uma biblioteca de especificação de *interface* para a transmissão de mensagens em um ambiente paralelo. Os dados são transferidos a partir de um endereçamento entre processos. Todos esses mecanismos tentam auxiliar o programador a desenvolver sistemas sem a preocupação de estabelecer comunicação e sincronização entre os processos de cada dispositivo.

Assim como a OpenMP e a MPI, o *MapReduce* é um modelo de programação que auxilia o desenvolvedor a implementar sistemas com transparência. Trata-se de uma camada entre a aplicação e o ambiente, capaz de abstrair ao desenvolvedor, a preocupação que o mesmo necessita ter ao trabalhar em um ambiente distribuído. Sua implementação utiliza o conceito de *cluster* de computadores, fornecendo um sistema escalar e com alto poder de processamento. Com este modelo é possível processar uma grande quantidade de dados em vários computadores. Todo o tratamento deve ser realizado pelo *MapReduce*, que irá manipular e gerenciar os dados em um conjunto de dispositivos. Sua principal característica é a capacidade de separar os dados e distribuí-los junto ao código que deverá ser executado entre os diversos nós do sistema.

Esse modelo possui raízes na programação funcional, utilizando o conceito de mapear e reduzir, característica comum do paradigma funcional. Este modelo foi desenvolvido por dois funcionários da Google, Jeffrey Dean e Sanjay Ghemawat, que o apresentaram na conferência OSDI'04 (*The Operating System Design and Implementation Conference 2004*). A principal preocupação de Dean e Ghemawat era encontrar uma possível maneira de paralelizar o processamento, distribuir os dados, manipular falhas e gerenciar o balanceamento de cargas no sistema. Neste documento, foi apresentado um sistema capaz de solucionar esses problemas relacionados ao processamento distribuído, operando em grandes *clusters*. Esse modelo foi denominado *MapReduce*, proveniente da junção do nome de duas funções, mapear (*map*) e reduzir (*reduce*). Inicialmente, o sistema foi desenvolvido utilizando a linguagem de

programação LISP (*List Processing*), devido a sua característica de adotar o paradigma funcional.

O modelo *MapReduce*, utiliza uma sequência de passos, como ilustrado na Figura 3.1. Dado um conjunto de dados, é possível separá-los em pequenas partes e distribuí-las entre os diversos nós de um *cluster*, que deverá ser especificado pelo programador. Essas distribuições são transferidas e organizadas em pares definidos por uma chave e um valor. Cada par deverá ser direcionado a um determinado nó. No exemplo da Figura 3.1, são transferidos documentos entre as diversas máquinas do sistema. Os dados distribuídos entre as máquinas são definidos por uma chave (*c1*) representada pelo identificador do documento e por um valor (*v1*). No caso da Figura 3.1, a chave é especificada pelo nome do arquivo enquanto o valor é determinado pelo seu conteúdo.

O documento é processado pelas máquinas que possuem a função *map* e encaminhado através de um par de chave-valor (*c2*, *v2*) para outro dispositivo, encarregado de processar a função *reduce*, cuja finalidade é combinar os valores recebidos. A saída (gerada pelo *node 05*) é definida por um par de chave-valor, especificado por um identificador e o resultado da combinação da saída de cada nó que executou a função *map*.

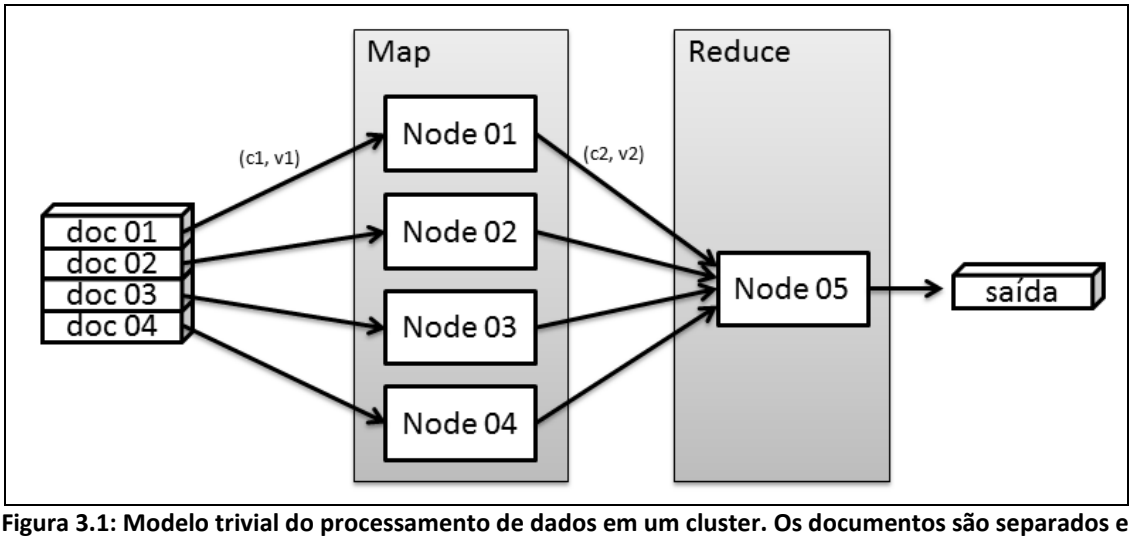

Figura 3.1: Modelo trivial do processamento de dados em um cluster. Os documentos são separados e distribuídos entre os diversos nós, que processam essas informações e enviam a um outro nó que deverá executar a função de combinação e retornar uma saída.

O exemplo apresentado nesta seção, demonstra um modelo trivial sobre a definição do *MapReduce*. Ao longo deste capitulo, serão apresentados novos conceitos com o intuito de amadurecer esse sistema, apresentando uma melhor definição do seu funcionamento, mostrando cada etapa desse processo, desde a separação dos dados até a combinação dos

resultados. A próxima abordagem trata-se do levantamento do conceito de programação funcional, dando ênfase às duas funções principais que resultaram no modelo abordado neste documento: *map* e *fold* (neste contexto, agrupar).

3.2 Paradigma da Programação Funcional

A programação funcional originou-se antes mesmo do início da era da informática, sendo criada sobre forte influência do cálculo lambda (cálculo-λ). Sua principal característica é o uso de funções de ordem superior[18] (THOMPSON, 1999, p. 155-156), utilizado para definir notações lógicas de funções matemáticas. O cálculo-λ foi criado em 1930 por Alonzo Church e Stephen C. Kleene (PETRICEK;SKEET, 2009, p. 21), com o intuito de determinar funções matemáticas através de uma notação. Eles desenvolveram uma solução para reduzir expressivamente cálculos abordando uma maneira sistemática para expressar fórmulas matemáticas, auxiliando seus estudos com o fundamento da matemática.

As linguagens de programação funcional utilizam artifícios influenciados pelo cálculo-λ para oferecer uma linguagem elegante e sucinta capaz de organizar algoritmos de forma legível. Com o avanço da tecnologia, a programação funcional perdeu espaço entre os programadores atuais, porém, esta visão vem se invertendo ao longo do tempo. Linguagens funcionais mostraram-se bastante eficientes para o processamento de grandes conjuntos de dados, melhorando drasticamente o desempenho de sistemas.

Uma das principais características das linguagens funcionais são as funções de alta ordem, que podem ser transmitidas via argumentos para outras funções, resultando em um alto nível de abstração. Esse mecanismo torna o código mais elegante e legível. Outra técnica bastante utilizada em linguagens funcionais é denominada *Currying*, cuja finalidade é receber múltiplos parâmetros em uma única função (THOMPSON, 1999, p. 185). A utilização dessa técnica, combinada ao uso de funções de alta ordem, trouxe uma maior flexibilidade às aplicações, que puderam aumentar seu desempenho e alcançar um nível de escalabilidade maior.

A programação funcional mostrou-se mais eficiente quando utilizada em sistemas paralelos, pois ela possui uma característica bastante importante na maioria das estruturas, a imutabilidade. Os programas podem ser executados em paralelo sem a preocupação de ocorrer

[18] O termo 'ordem superior' é designado para informar o uso de passagem de funções como argumento de outra função, ou seja, pode-se passar como valor de uma função, outra função.

problemas, como no caso das condições de corrida e a criação de seções críticas (PETRICEK;SKEET, 2009, p. 6). O conceito de imutabilidade dos valores, trouxe grandes benefícios ao desenvolvimento do modelo *MapReduce*, pois é possível executar funções idênticas em grandes *clusters* de máquinas sem a preocupação do compartilhamento de informações entre os processos.

A primeira linguagem que se popularizou entre os cientistas, matemáticos e desenvolvedores, foi a LISP, criada por John McCarthy em 1958, uma linguagem flexível e baseada no cálculo-λ (PETRICEK;SKEET, 2009, p. 21). O uso dessa linguagem trouxe grandes benefícios à computação atual. A partir dela surgiram grandes idéias tais como estruturas de dados e a coleta de lixo (*Carbage Collection*). Outras linguagens funcionais também contribuíram para a evolução da computação como as linguagens ML, Haskell, F# e a OCaml.

3.2.1 Funções generalizadoras: Mapeamento e Redução

O uso de funções generalizadoras (THOMPSON, 1999, p. 152; DE SÁ, 2006, p. 146) na programação funcional, desempenhou um papel importante na legibilidade e desempenho dos algoritmos que utilizam esse paradigma. Esse tipo de característica é um padrão utilizado para auxiliar o caso da reusabilidade de código, onde o desenvolvedor pode implementar sistemas utilizando técnicas de funções de alta ordem. Devido a essa particularidade, é possível que linguagens de programação funcional possam generalizar funções em qualquer situação.

A principal utilização das funções generalizadoras é a manipulação de estruturas de dados, onde é possível desenvolver funções que definam resultados sobre essas estruturas, gerando novos valores a partir dos dados recebidos. As funções mais populares utilizadas em algoritmos da linguagem funcional são conhecidas como *map* e *fold*. Essas permitem que os dados possam ser manipulados, gerando uma nova informação sem alterar os dados de entrada.

A função de mapeamento consiste em modificar os valores de uma lista, onde cada componente deverá ser modificado, resultando em uma nova lista com novos componentes. Por convenção, o código de uma função de mapeamento será apresentado utilizando a notação da linguagem Haskell (DE SÁ; DA SILVA, 2006, p. 147-148):

```
#1    maiusc :: String -> String
#2    maiusc :: [] -> []
#3    maiusc :: (a:b) toUpper a: maiusc b
#4
#5    ret_maiusc :: [String] -> [String]
#6    ret_maiusc :: [] -> []
#7    ret_maiusc :: (a:b) maiusc a: ret_maiusc b
```

A função *map* é definida como um método que recebe e retorna uma lista de *strings* (linha 1). Caso receba alguma lista vazia, então o resultado também deverá retornar uma lista vazia. A notação *a:b* afirma que será realizada a função "*maiusc*" para o primeiro elemento da lista e novamente os demais elementos restantes, serão enviados como parâmetros para a função "ret*_maiusc*". Na função "*maiusc*", são verificadas todas as letras pertencentes à cadeia de caracteres, e então executada a função "*toUpper*".

A chamada a função é realizada através de um *console* (MEIRA, 1988, p. 9), representado por uma seta (->), que recebe uma lista *strings* para executar o procedimento "*maiusc*":

```
-> ret_maiusc ["Celso", "Haskell", "Mapeamento"]
```

Pode-se verificar resultado da função, através da saída do nosso console fictício:

```
-> maiusc ["CELSO", "HASKELL", "MAPEAMENTO"]
```

A função de agrupamento incide em combinar os valores de uma determinada lista. O código da função de agregação é apresentado utilizando a notação da linguagem Haskell (DE SÁ; DA SILVA. 2006. p. 149):

```
#1    concat :: [String] -> String
#2    concat :: [] -> []
#3    concat :: (a:b) a ++ (concat b)
```

A entrada da função *fold*, consiste em uma lista de *strings* que deverá ser concatenada e retornada como uma única *string*. Cada valor da lista deverá ser concatenado através da notação ++. A função de agrupamento é executada pela seguinte chamada:

```
-> concat ["exemplo ", "de função ", "fold."]
```

O resultado gerado pela função *fold* é apresentado da seguinte forma:

```
-> "exemplo de função fold."
```

A utilização do conceito de funções generalizadoras, como no caso das funções de mapeamento e agrupamento, influenciou bastante no desenvolvimento do *MapReduce*. Trata-se de simples processos que dão suporte ao paralelismo e à escalabilidade devido às particularidades adotadas pelo paradigma funcional. A utilização dos métodos *map* e *fold* atuando em conjunto em diversos sistemas, geram grandes benefícios para a organização desse modelo auxiliando no processamento de dados, sem a preocupação da concorrência de informações.

3.3 Combinando Map e Fold

Uma das técnicas aplicadas no *MapReduce* bastante úteis para a estruturação do modelo é a combinação das funções generalizadoras, ilustrada na Figura 3.2. Dado um conjunto de informações (representado por uma circunferência) é possível separá-las em várias partes e processá-las em paralelo, a partir de uma função de mapeamento (definida pela letra m). O resultado de cada *map* (representado por um triângulo) servirá de entrada para cada elemento *fold* (definida pela letra f), que também receberá como parâmetro o resultado da função de combinação anterior (representado pelo quadrado). No caso da primeira execução *fold*, os valores iniciais (representado pelo primeiro quadrado, da esquerda para a direita) deverão ser um conjunto de dados vazio, por questões óbvias.

As utilizações das funções de alta ordem podem ser verificadas da seguinte forma: Suponha que em nosso problema exista uma lista de valores. A função *map* captura um único valor dessa lista e utiliza-o como argumento para o resultado da função, que é retornado como um valor arbitrário (apenas um valor intermediário). A saída da função é definida de acordo com as regras especificadas pelo programador que desenvolveu o algoritmo. A função *fold* utiliza os valores intermediários, retornados pelo *map* e utiliza-os como argumento para a sua função. Vale ressaltar que a função de agrupamento utiliza dois argumentos, um valor inicial e o valor intermediário. O resultado desta função retorna também um valor intermediário, que servirá de argumento para a próxima etapa.

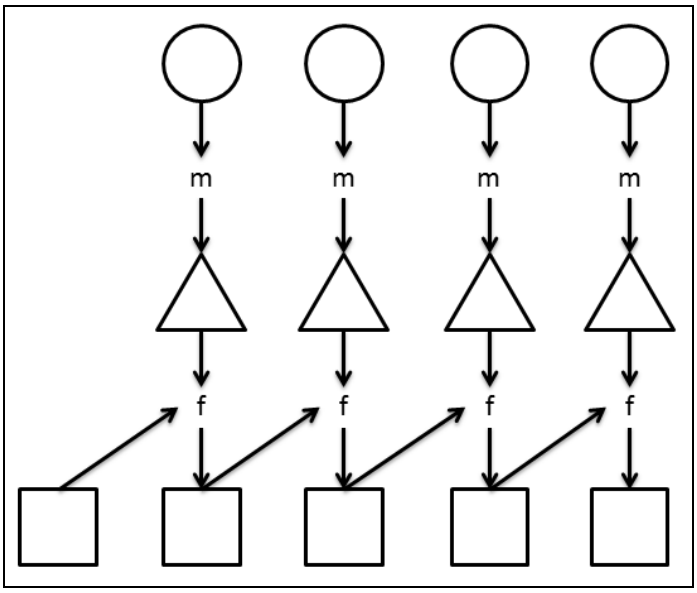

Figura 3.2: Combinação das funções *map* (representada pela função m) e *fold* (representada pela função f).

É notável que este tipo de implementação possua total influência do conceito ”dividir para conquistar”. Na função *map* cada valor de uma determinada lista é separado de forma independente, para que possa ser executado em paralelo. Caso nosso exemplo utilize um *cluster*, as tarefas podem ser divididas e distribuídas entre cada um dos nós do sistema. No caso da função *fold* existe uma restrição quanto à dependência das atividades. Para ele é necessário obter a saída de outra função. Porém, essa dependência não é tão critica como se espera, na medida em que os elementos da lista são separados, a função de agrupamento também é executada em paralelo.

A combinação das funções generalizadoras, definem de forma superficial o conceito do *MapReduce*. Dado um *cluster*, um conjunto de máquinas deverá realizar a tarefa *map*, enquanto os demais executam a tarefa *reduce*, que são similares à função mapeamento e agrupamento da programação funcional. O *framework* utiliza a execução dessas atividades forma paralela, garantindo o desempenho do sistema.

3.4 Mappers e Reducers

Até o momento foram apresentados somente conceitos do *MapReduce*. Esta seção define a estruturação deste modelo, onde cada máquina que compõe este sistema realiza uma atividade específica.

Como já visto no capitulo anterior, um *cluster* utiliza um conjunto de máquinas para processar o sistema de forma transparente. O modelo *MapReduce* também utiliza este conceito de transparência. Cada dispositivo é responsável por uma atividade dentro do sistema. No caso dos dispositivos responsáveis pela função *map*, estes são denominados *mappers* enquanto os que executam a função *reduce* são conhecidos como *reducers* (LIN, 2010, p. 22). Para os respectivos agentes, as assinaturas de cada função são especificadas com as seguintes assinaturas (DEAN; GHEMAWAT, 2004, p 2):

```
map: (c1, v1) -> (c2, v2)
reduce: (c2, list(v2)) -> (c3, v3)
```

Por convenção uma lista de valores será representada pelo uso da marcação '*list (...)*', que deverá conter como argumento da função o tipo dos valores. A saída da função será especificada pela seta (->), onde as informações anteriores a ela informam os argumentos de entrada da função, enquanto os dados após a seta especificam o resultado gerado pela mesma.

Para executar um *mapper*, é necessário utilizar dois argumentos, uma chave (especificada pelo componente *c1*) e o seu respectivo valor (*v1*). O resultado dessa função é gerado como um dado intermediário, com uma chave e um valor (*c2, v2*). Esse tipo de resultado é conhecido como par de chave-valor, que é sempre apresentado como uma indexação, ou seja, para cada valor, existe uma determinada chave. Os dados utilizados durante a função *map* são especificados pelo programador, que podem ser representados por qualquer tipo, tais como variáveis de inteiros, pontos flutuantes e *strings*, ou até mesmo estruturas complexas como listas e *arrays*. Outras estruturas também podem ser utilizadas dentro do modelo *MapReduce*, como no caso das páginas *web* e dos grafos. Exemplos da utilização dessas estruturas podem ser encontradas em Lin (2010, p. 96 - 102), nos algoritmos de *pagerank* e Dijkstra.

O *MapReduce*, utiliza uma camada implícita entre as fases *map* e *reduce* para ordenar e agrupar os valores que possuem a mesma chave em comum, dessa forma, um conjunto de valores é agrupado para uma única chave. Os *reducers* recebem como argumentos, a saída dos *mappers*, para cada chave intermediaria (*c2*) existe um conjunto de valores intermediários (*list (v2)*), ou seja, uma lista de valores é relacionada a uma chave, que será designada a um *reducer*. A saída gerada deste, resulta em um novo par de chave-valor (*c3, v3*), cada resultado é armazenado no sistema de arquivos distribuídos. Caso exista uma quantidade *R* de *reducers*, um conjunto de *R* arquivos deverá ser cadastrado. As saídas geradas pelos agentes que

utilizam a função *reduce*, nem sempre são transformadas em um único arquivo. Porém, nada impede que eles sejam processados novamente por outra tarefa *MapReduce*, neste caso, um outro *reducer* poderia se encarregar de unir as *R* saídas. A Figura 3.3 ilustra toda a organização dos *mappers* e *reducers*.

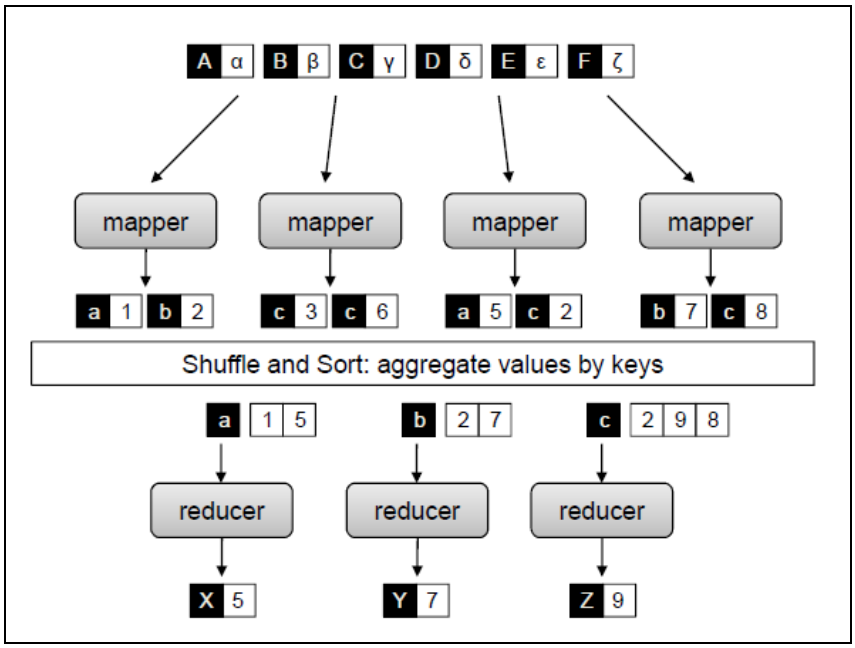

Figura 3.3: Representação do modelo *MapReduce*. A camada 'Shuffle and Sort' indica que os pares de chave-valor serão agrupados de acordo com as chaves em comum, estes servirão de entrada para os Reducers, que deverá gerar uma nova chave com um novo valor de acordo com a função especificada pelo programador. (LIN; DYER, 2010, p. 23).

3.5 Word Count MapReduce

Nesta seção será abordada a execução de um modelo *MapReduce* utilizando um algoritmo de contagem de palavras, que deverá retornar a quantidade de palavras existente em um conjunto de textos. Para os *mappers*, as entradas são especificadas por um par de chave-valor, onde a chave é definida pelo documento e o valor pelo conteúdo do documento. Os *reducers* possuem a capacidade de agrupar cada incidência de palavra de acordo com uma chave como demonstrado pelo algoritmo:

```
#1    map (String chave, String valor) {
#2          foreach L in valor;
#3                gerarPalavraIntermediaria (L, "1")
#4    }
```

```
#5    reduce (String chave, List valor) {
#6          int resultado = 0;
#7          foreach v in valor;
#8                resultado  += (int) v;
#9          emitirResultado( (String) resultado);
#10   }
```

A função '*gerarPalavraIntermediaria*' é utilizada para gerar um novo par de chave-valor. Neste caso, para cada palavra, a função *map* deverá gerar o valor 1 (um), simbolizando que foi encontrada uma palavra. É possível que este tipo de processamento possa ser realizado em paralelo. Dado um *cluster* de máquinas, é possível que cada documento possa ser separado em várias partes, que serão enviadas para os *mappers*, que deverão emitir um novo par de chave-valor.

Para os *reducers*, a única tarefa a ser realizada é a combinação de todos os resultados gerados pela função de mapeamento. Neste caso, os resultados são armazenados de uma maneira que essa informação seja compartilhada entre todos os dispositivos pertencentes ao sistema, como no caso de um arquivo alocado em um ambiente distribuído. O resultado final é apresentado pela função '*emitirResultado*' que retorna o valor de toda a combinação feita pelo *reducer*. A Figura 3.4 ilustra todo o processo do exemplo citado.

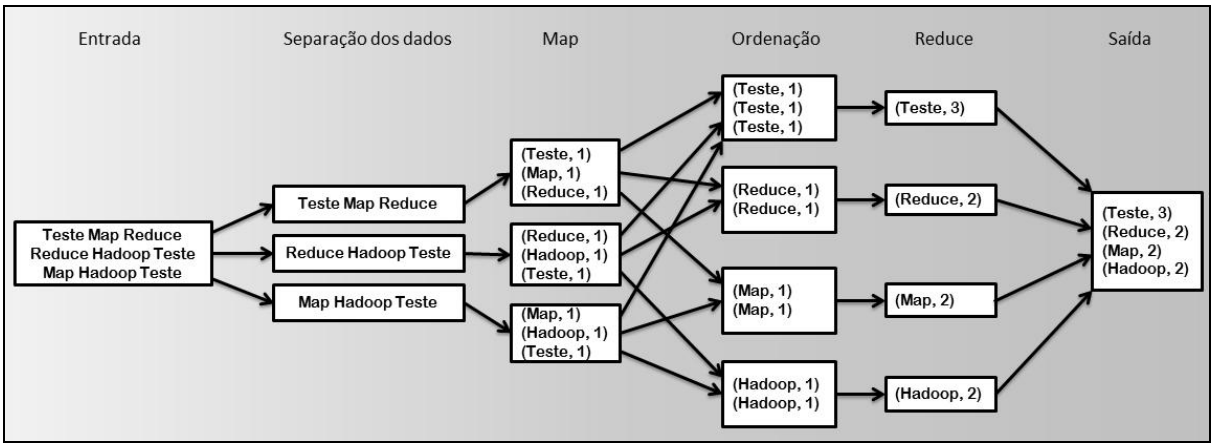

Figura 3.4: Diagrama de execução do algoritmo de contagem de palavras.

No exemplo da Figura 3.4, o fluxo de execução do modelo *MapReduce* é definido por etapas. Na separação dos dados, a entrada é dividida em partes, de acordo com as especificações impostas pelo programador. Para o exemplo citado, as informações foram separadas de acordo com a quebra de linha. Sendo assim, cada nó *mapper* deverá executar uma pequena parte do arquivo. Cada instância da função *map* deverá emitir um par de chave-valor, onde a chave será especificada pela palavra e o valor pela quantidade que ela aparece

no texto (neste caso um única vez). A etapa de ordenação[19] é definida a partir da semelhança entre as chaves, ou seja, cada instância deverá armazenar um conjunto de pares que possuam a mesma chave em comum. Por fim, cada par será combinado dentro dos *reducers*, que deverão retornar a quantidade de palavras contidas no texto. Neste exemplo, o resultado final foi combinado em uma única saída, porém, nada impede que esta função resulte em um conjunto de *R* saídas definidas pelos *R reducers*.

O exemplo da contagem de palavras pode resolver um problema conforme a seguinte situação. Suponha que em um ambiente distribuído existem 10 bilhões de documentos, e cada um desses possui cerca de 50 KB. Para esse exemplo, um sistema precisa processar aproximadamente 500 *terabytes* de informação. Com uma quantidade elevada de dados, um dispositivo poderia demorar horas processando uma quantidade imensa de informações, sobrecarregando o sistema e reduzindo o poder de resposta da aplicação, algo totalmente relevante para atender às necessidades dos usuários.

Utilizando o modelo de programação paralela, o sistema seria capaz de executar o mesmo cálculo para cada página da *web* de forma rápida e eficiente, utilizando conceitos complexos de bloqueios de seção crítica entre outros artifícios. Ainda assim, o sistema estaria sobrecarregado para executar uma quantidade significativa de processos. Devido a utilização dos conceitos da programação funcional, o *MapReduce* não necessita fazer o uso de artifícios tais como bloqueios, nem garantir algum controle de concorrência para executar o processamento em paralelo, resultando em códigos mais limpos e de fácil legibilidade processando dados com maior velocidade.

3.6 Execução do modelo em um Sistema Distribuído

Em um sistema distribuído, diversas máquinas executam várias tarefas simultaneamente. Utilizando a técnica *MapReduce*, o sistema adquire uma capacidade de melhor escolha para a execução dos seus processos. A arquitetura desse modelo está baseada na divisão hierárquica dentro de um *cluster* de computadores, que adotam um modelo de divisão de atividades onde uma máquina deverá enviar solicitações para a execução de processos enquanto as demais executam essas requisições.

[19] Esta função será explicada posteriormente quando for apresentado o conceito de *partitioners* e *combiners*.

As máquinas conhecidas como *masters* possuem a capacidade de separar e direcionar os processos dentro de um sistema, garantindo que uma solicitação seja direcionada a uma máquina arbitrária, denominada *worker*. Neste tipo de sistema, a divisão hierárquica das funções é garantida através de uma organização, onde ao menos uma máquina *master* gerencia um grupo de *workers*, podendo esses serem *mappers* ou *reducers*, conforme a função obtida a partir de uma solicitação do *master*.

A execução de um programa *MapReduce* baseia-se em separar os dados em pequenas partes para que possa ser executado por vários nós de um *cluster*. Porém, em alguns casos estas divisões extrapolam a quantidade de nós existentes no sistema. Para evitar uma sobrecarga dessas atividades, alguns programas utilizam técnicas de escalonamento para que os dados possam ser processados de forma eficiente. Uma fila de atividades é utilizada, e assim que um dos *workers* torna-se disponível, o sistema transfere uma tarefa para este nó.

Outro fator que torna o *MapReduce* um modelo ideal para processamento de grandes quantidades de dados, é a capacidade de transferir o código para os nós que armazenam estas informações. Dessa forma, os dados são processados em sua unidade local, e caso um processo necessite das informações de um nó que esteja executando alguma funcionalidade, o sistema replica estes dados (através da rede) para nós ociosos a fim de executar as atividades pendentes. Um grande avanço para os sistemas que utilizam este modelo é o fato de escolher computadores que se encontram no mesmo *datacenter*, para que não seja preciso a utilização de uma maior largura de banda.

O processo de sincronização auxilia na confiabilidade do sistema, pois se trata de um compartilhamento de informações. Tanto *mappers* quanto *reducers* precisam garantir que os dados estejam sincronizados de acordo com suas referências através da camada "*shuffle and sort*". Esse tipo de organização necessita que os dados sejam copiados e transferidos pela *web*, e no caso de utilizarem *M mappers* e *R reducers*, a quantidade de dados trafegados pela rede deverá ser o produto desses dois agentes, ou seja, *M x R*. É possível verificar que os *reducers* necessitam dos resultados gerados pelos *mappers*. Neste caso, alguns nós que utilizam a função *reduce* deverão esperar por uma resposta da função *map*. O Hadoop[20] desenvolveu uma solução para evitar a ociosidade entre as máquinas. Quando um *mapper* emite alguma resposta, essa é automaticamente transferida para o *reducer*, ou seja, à medida que as respostas são emitidas, elas são replicadas e transferidas imediatamente para que um nó possa processá-la.

[20] http://hadoop.apache.org/.

As práticas de divisão e conquista são bem aplicadas neste modelo, é notável o ganho quanto ao desempenho do sistema. Porém ainda existe um desafio para a organização dos dados processados. Com o modelo organizado desta forma, a criação dos arquivos pelo *reducer* poderia gerar um alto índice de informações descentralizadas. Em cada *cluster* de máquinas poderia ser encontrado uma parte das informações necessárias para o usuário, contribuindo para o aumento do tráfego de dados pela rede, diminuindo o desempenho do sistema.

Como uma forma de solucionar esse problema, foram criados dois elementos cuja finalidade é organizar e direcionar as informações que possuem alguma relação. O primeiro agente é conhecido como combinador (*combiner*), sua função é realizar o agrupamento dos valores que possuem a mesma chave em comum. Este tipo de implementação auxilia o desempenho do sistema, pois evita que mais dados sejam trafegados entre os diversos *clusters*. Trata-se basicamente de um agrupamento, realizado localmente, antes mesmo das saídas dos *mappers*.

O segundo elemento que trouxe benefícios ao desempenho do modelo *MapReduce* é denominado particionador (*partitioner*), este por sua vez, garante a classificação dos pares chave-valor em diversas máquinas. Dessa forma, os dados que possuem a mesma chave são alocados em um único local para evitar fragmentação das informações. Dessa forma, toda informação que possuir uma chave de mesmo valor (geralmente essa comparação é feita a partir de uma função *hash*) será direcionada a um determinado computador, gerando um conjunto de dados que possuem alguma semelhança (na maioria dos casos), direcionado a um determinado *reducer*.

A Figura 3.5 apresenta uma exemplificação de como atuam esses dois procedimentos. Os *combiners* realizam um agrupamento dos valores gerados pelo mapper. Dessa forma, evita-se a transmissão desnecessária de informações, melhorando o desempenho do sistema, pois a combinação é realizada antes mesmo da informação ser enviada a um *reducer*. No caso dos partitioners, cada par de chave-valor deverá ser redirecionado a um determinado reducer, ou seja, sua principal funcionalidade é direcionar cada informação à um determinado nó, de acordo com as especificações do programa.

Os Sistemas distribuídos foram mostrados apenas como uma técnica capaz de solucionar diversos casos de processamentos de dados. Em ambientes cuja comunicação ocorre através da comutação de mensagem pela *web*, erros são bastante frequentes, podendo

ocorrer falhas na conectividade ou em um *hardware* de um nó, ou seja, nenhum sistema está livre de problemas.

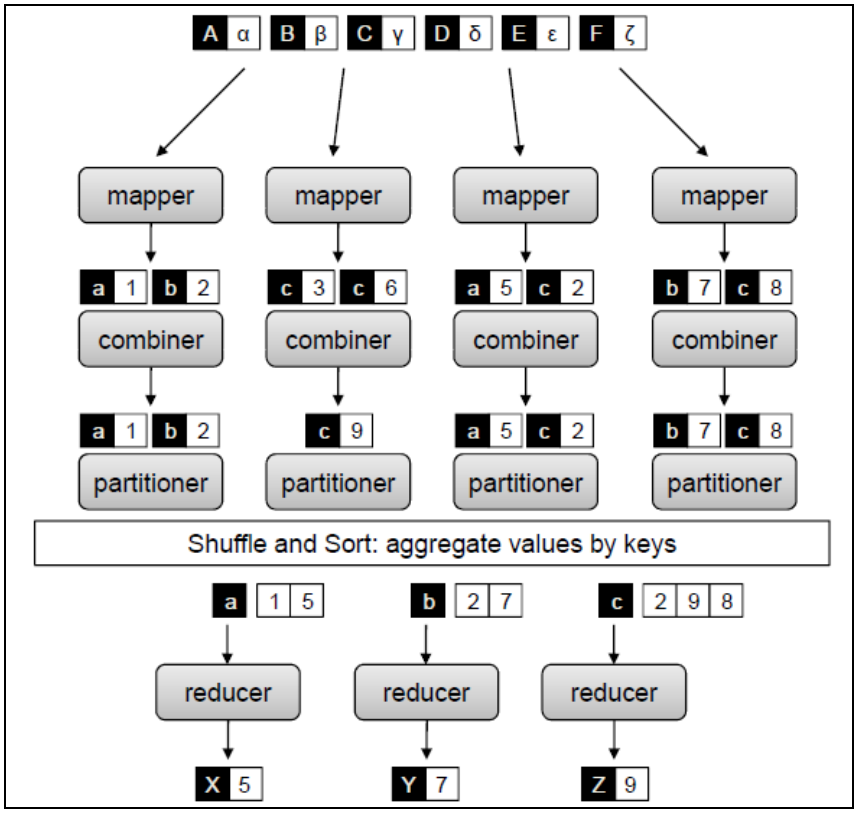

Figura 3.5: Modelo de organização do *MapReduce* apresentando os procedimentos realizados pelos Partitioners e Combiners. A camada shuffle and Sort, trata de agrupar todos os valores que possuem uma chave em comum. (LIN; DYER, 2010, p. 30).

O *master* verifica periodicamente se os *workers* estão conectados à rede através do comando *ping*[21], caso nenhuma resposta seja recebida, então o nó recebe uma marcação informando que o mesmo encontra-se inativo. Se um determinado nó inacessível estivesse realizando alguma tarefa *map* ou *reduce*, então esta atividade torna-se elegível para o escalonador e poderá ser executada novamente por outra máquina. Todos os demais *workers* são informados que ocorreu uma falha em um determinado nó e deverão apontar suas referências de um conjunto de dados especifico, ou seja, ao novo nó que executou a tarefa da máquina defeituosa. O modelo *MapReduce* é um sistema bastante robusto e tolerante a falhas,

[21] O commando ping é utilizado para verifica se as máquinas estão conectadas na rede, este utiliza o protocolo ICMP que envia várias mensagens para um determinado computador. A sintaxe para a execução deste comando é realizada pelo comando 'ping endereço', onde o endereço especifica o IP da máquina que receberá a mensagem.

mesmo que um grupo de máquina estejam inacessíveis, o *master* detecta falhas no sistema e reexecuta todas as tarefas incompletas.

É notável que o sistema tolerante a falhas somente verifique problemas em nós *workers*. Em uma situação onde a falha ocorre no nó *master*, o sistema poderá parar. No entanto, uma cópia do *master* é iniciada e o trabalho poderá ser reexecutado a partir de um *checkpoint*. A Figura 3.6 ilustra uma distribuição hierárquica dos nós em um sistema, o programa do usuário executa o nó *master* que transfere as funções aos respectivos *workers*, os *mappers* e os *reducers* a partir de uma biblioteca *MapReduce*. Nota-se que toda saída da função *map* é armazenada em um arquivo localmente.

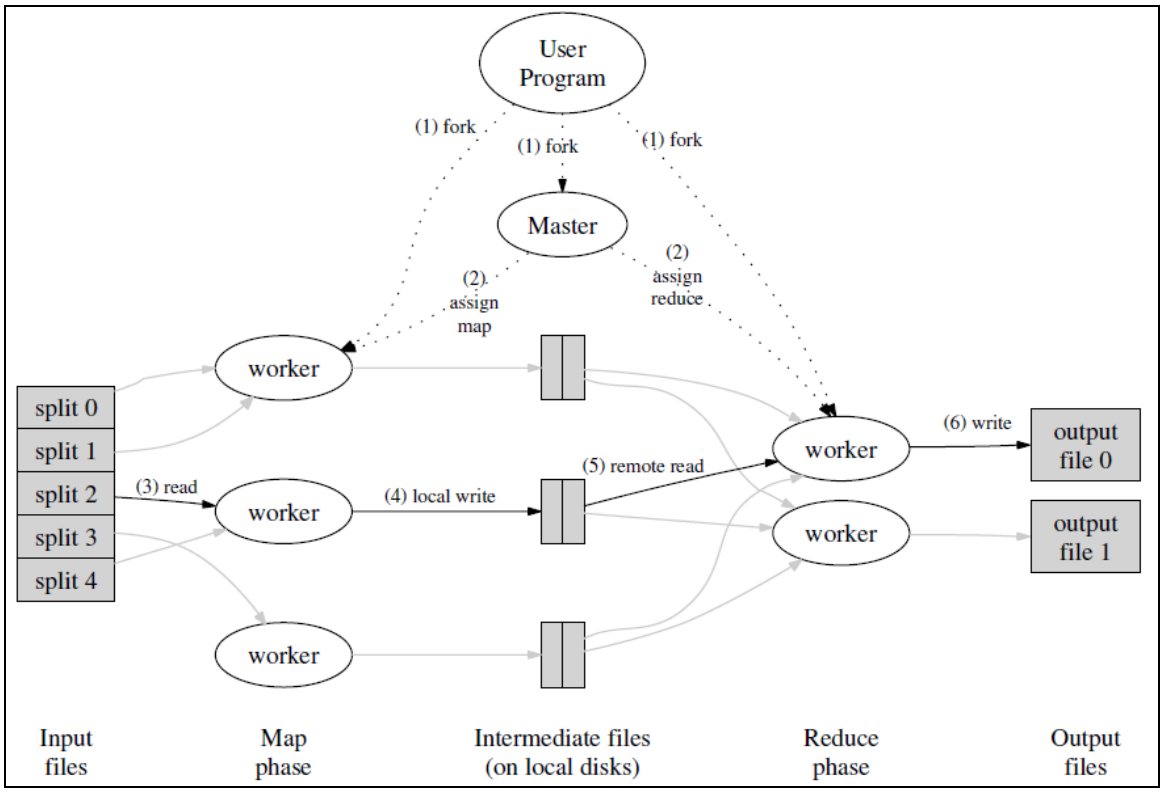

Figura 3.6: Distribuição de um modelo *MapReduce*, demonstrando todas os passos do processo. (DEAN; GHEMAWAT, 2004, p 3)

O exemplo da Figura 3.6 trata-se de um diagrama idealizado e utilizado por Dean e Ghemawat. O programa do usuário inicia todo o processo executando a função *MapReduce*. Primeiramente, a biblioteca do sistema separa os arquivos em *M* partes de tamanhos variados

entre 16 e 64 MB. Após a separação dos arquivos, o programa do usuário inicia várias cópias do processo em vários nós de um *cluster* (executado na marcação 1 da Figura 3.6)[22].

O dispositivo que recebe a cópia do sistema *master*, tem como principal função associar as funções *map* e *reduce* aos nós *workers* (2). Assim como os arquivos separados, existem M nós *mappers* e R *reducers*. A associação das funções é realizada somente aos nós que se encontram ociosos. Cada um dos M arquivos deverá estar associado aos *workers* que receberam a tarefa de executar a função *map* (3). A saída desses processos estará armazenada em um *buffer* de memória e assim que concluída, deverão ser escritas em disco local (4). Após serem armazenados localmente, os *Mappers* passam a localização exata dos arquivos para o *master* que deverá informar aos *reducers* onde procurar as informações necessárias para a execução do processo.

Quando um *reducer* é notificado sobre a localização de um determinado arquivo localizado em um *mapper*, este executa uma chamada RPC para copiar as informações para a memória (5), as informações serão copiadas, classificadas e agrupadas de acordo com os dados que possuem a mesma chave em comum. Caso a quantidade de dados seja maior que o esperado, e o nó não possua capacidade de processá-lo, então deverá ser necessária a utilização de um recurso externo para executar o processo de classificação e agrupamento dos dados. Após todos os dados serem copiados e processados, os resultados da operação *reduce* deverão gerar um arquivo e armazená-lo localmente em um nó *reducer* (6).

Após todas as funções serem executadas, o *master* é notificado e envia uma mensagem ao programa do usuário informando que o processo foi concluído conforme solicitado. O resultado gerado pelo programa *MapReduce* apresenta como saída um conjunto de R arquivos, um para cada *Reducer*. Cada saída poderá ser processada novamente por um novo programa *MapReduce* ou armazenada em um sistema de arquivos distribuído, conforme as regras especificadas pelo programador.

3.7 Considerações Finais

O *MapReduce* pode ser definido de diversas formas distintas: pode referir-se a um modelo de programação desenvolvido por Dean e Ghemawat, ou ser visto como um

[22] Na Figura 3.5 foi apresentada a criação dessas copias utilizando o comando *fork*, porém a utilização desse comando só ocorre em sistemas locais, não podendo ser executado em ambientes distribuídos. Trata-se apenas de um conceito onde auxilia no entendimento das etapas do processo *MapReduce*.

framework em execução que manipula e gerencia a execução de alguns programas específicos, ou até mesmo ser atribuído a uma implementação especifica que utiliza o modelo de programação junto a um *framework*, tais como a implementação realizada pela Google ou o Hadoop, sistemas que estão se popularizando cada vez mais devido ao seu alto poder de escalabilidade e desempenho.

Os conceitos apresentados anteriormente expressam corretamente o que vem a ser este modelo. Porém o *MapReduce* vai além destas definições. Trata-se de uma ideologia capaz de classificar os dados, processá-los e indexar os resultados de forma rápida e eficiente. Podendo ser aplicado a inúmeros problemas, onde cada entidade receberá uma determinada função para aplicá-la junto à informação que recebera, com o mínimo de tráfego na rede. Grande parte do processamento é executado localmente. As tarefas são divididas em grupos de M *mappers* e R *reducers*, para o *master* a quantidade de escalonamento a ser realizado é na ordem de O (M + R) e mantém O (M * R) estados em memória (DEAN; GHEMAWAT. 2004. p. 5). A quantidade de espaço armazenado na memória é mínima, aproximadamente 1 (um) *byte* por *worker* (no modelo do Google File System, que será abordado na próxima seção).

O *MapReduce* se popularizou entre os diversos sistemas da atualidade, empresas utilizam o conceito deste modelo para garantir desempenho, escalabilidade e agilidade no desenvolvimento de aplicações para sistemas fracamente acoplados. Diversas empresas como a Microsoft, Amazon e a Yahoo utilizam este modelo para o processamento de grandes quantidades de dados e ferramentas como a API Google garantem a abstração das máquinas para os programadores, mesmo os que não possuem experiências com ambientes distribuídos podem com facilidade implementar diversas aplicações. Outro fator importante é o processamento dos dados em máquinas locais, esse artifício auxilia o desempenho de diversas funções ao longo do sistema e permite um melhor balanceamento de carga, pois gerencia os dados associando-os a máquinas que aguardam por algum processo. Dessa forma, vários processos podem ser executados paralelamente sem a preocupação de sobrecarga de maquinas.

Até o presente momento, foram apresentados somente exemplos de algoritmos fictícios para a resolução de simples problemas, a próxima seção deverá abordar um estudo de caso sobre diversas ferramentas utilizadas por grandes empresas que disponibilizam seus serviços através das nuvens com alto desempenho e integridade das informações.

4 ESTUDO DE CASO

Até o presente momento foram apresentados conceitos do que vêm a ser e como são processados os dados do *MapReduce*. Nesta seção serão abordados sistemas que utilizam este modelo para o processamento de grandes conjuntos de dados com alto desempenho. O desenvolvimento de sistemas que utilizam essa forma de implementação está aumentando cada vez mais todos os dias, empresas como a Facebook, LinkedIn e o Twitter, adotaram a implementação de ferramentas que abordam este conceito com o intuito de melhorar o poder de resposta. Para estes sistemas é necessário que as informações sejam lidas e escritas de forma rápida e segura, garantindo a confiabilidade do sistema.

4.1 Google File System - GFS

O Google File System trata-se de um sistema de arquivos distribuídos proveniente da Google. Inicialmente foi projetado para ser um sistema utilizado para atender às necessidades relacionadas ao armazenamento e processamento de grandes conjuntos de dados. É utilizado para diversas finalidades e desenvolvido em *clusters* que alocam centenas ou até milhares de nós. O maior já implementado, utiliza um conjunto de 1000 nós, com uma capacidade acima de 300 *terabytes*, podendo ser acessado por centenas de clientes em diferentes máquinas simultaneamente e em bases continuas (GHEMAWAT, 2003, p. 1). Esse sistema visa atender às demandas de um problema que vem crescendo constantemente: a capacidade de processamento de dados em larga escala, garantindo a escalabilidade, confiabilidade e disponibilidade das informações.

Uma das facilidades do GFS está na sua *interface* simples e intuitiva, semelhante aos sistemas de arquivos comuns. Sendo organizados de forma hierárquica através de diretórios e especificados por identificadores, como os nomes dos arquivos e o caminho onde estão alocados. Além das questões de acesso, a *interface* suporta diversas operações, como a abertura de arquivos, leitura, escrita e remoção do mesmo.

A principal motivação do GFS é garantir a segurança das informações contidas no sistema. Falhas podem ocorrer a qualquer momento, portanto é necessário garantir um sistema que possa solucionar esses problemas, utilizando estratégias como a replicações de dados e o monitoramento dos dispositivos que compõem o sistema. Outra preocupação do sistema de

arquivos da Google é garantir a manipulação e o gerenciamento de grandes conjuntos de dados, visando sempre que esses arquivos estão sendo constantemente acessados. Dessa forma é necessário criar um sistema que seja rápido o suficiente para a leitura dessas informações. Os dados acessados podem ser alterados de forma concorrente, ou seja, vários usuários podem alterar os mesmos elementos através do mecanismo de *record append*, ao mesmo tempo que garante também a atomicidade do arquivo.

A arquitetura do sistema GFS, consiste na mesma idéia do *MapReduce*, onde existem nós separados por uma hierarquia de funcionalidade. O *master* (assim mesmo chamado) é designado para manipular todos os demais nós (*workers*), associando a cada máquina uma determinada atividade a ser realizada. Esses são conhecidos como *chunkservers* e acessados por múltiplos clientes.

Assim como no *MapReduce*, os dados são separados em blocos, denominados *chunks*. Cada bloco possui pequenas informações sobre a sua criação, conhecidas como manipuladores de blocos (*chunk handle*), esses elementos são imutáveis e únicos para o sistema. Cada *chunk* é armazenado nos *chunkservers* como um arquivo e possui um tamanho fixo de 64 MB (por padrão), podendo ser alterado caso seja necessário, para evitar a fragmentação interna[23]. A grande vantagem de separar os dados em blocos ocorre devido às questões de desempenho. Um cliente não necessita solicitar escrita e leitura de arquivos ao *master*, essas ações podem ser realizadas diretamente ao *chunk*, evitando uma grande sobrecarga do servidor central. Esse procedimento será explicado futuramente nesta seção. As informações sobre cada *chunk* é armazenada pelo *master*, que replica os blocos por todo o sistema (por padrão, são replicados para outros três nós do *cluster*).

A principal função do *master* é garantir o controle do sistema. Para isso, é necessário que ele obtenha informações de todos os *chunkservers*. Essas informações são conhecidas como metadados, que possui dados relacionados ao arquivo, a localização de cada *chunk* e também o *namespace*[24]. As informações referentes ao mapeamento dos dados e a sua localização, são armazenadas em *logs* pelo *master*. Dessa forma, dados sobre os *chunkservers* são armazenados em locais separados, para que as informações possam continuar persistentes em caso de falha do nó *master*. Além da segurança, outra vantagem do uso de *logs* dentro do sistema, ocorre por questões de desempenho. É possível que o *master* possa replicar alguns

[23] A fragmentação interna consiste na perda de espaço em blocos de tamanho fixo. A fragmentação ocorre quando os espaços dos blocos não são preenchidos por completo.

[24] Um *namespace* contém informações relevantes sobre um determinado arquivo, como o seu identificador, seu caminho e outras demais informações necessárias para garantir que esse seja um arquivo único dentro do sistema. Assim como nos sistemas de arquivos comuns.

chunks para diminuir a sobrecarga dos *chunkservers*, além de garantir a limpeza do sistema, utilizando o mecanismo de *garbage collection*.

O Google File System utiliza diversas estratégias de desempenho. Porém a utilização de alguns artifícios para essa finalidade, como no caso da memoria *cache*, são raramente utilizadas. Nesse caso, a utilização desse mecanismo só é valida no armazenamento dos metadados. Um *chunkserver* não necessita colocar os arquivos em *cache*, uma vez que esses estão alocados em um disco local. Por outro lado os clientes também não precisam armazenar em *cache* esses arquivos, devido à complexidade existente na alocação de grandes conjuntos de dados em máquinas cliente. A utilização do cache para armazenar informações importantes ao invés de incluir dados, trouxe simplicidade ao modelo. Devido a esse artificio, é possível que o nó *master* possa executar suas tarefas sem ser sobrecarregado por várias solicitações dos clientes. Dessa forma, evita a criação de um gargalo, pois clientes não lêem ou escrevem através de requisições ao *master*, apenas solicitam qual a localização de um determinado *chunk*. Todas as alterações de uma determinada informação são realizadas diretamente entre o cliente e o *chunkserver*.

A Figura 4.1 apresenta o modelo de funcionamento do GFS (GHEMAWAT; LEUNG; GOBIOFF, 2003, p. 3). Primeiramente o cliente solicita ao *master* a localização do *chunk*, enviando o nome e o índice do arquivo. Em seguida a aplicação recebe como resposta, o *chunk handle* e a localização dos *chunks* no sistema. O cliente armazena essas informações em *cache*, utilizando como índice os parâmetros utilizados na solicitação realizada inicialmente ao *master*. A partir desse momento toda a comunicação é realizada com o *chunkserver*, que recebe requisições do cliente, sendo essas o *chunk hadle* e o índice dentro do *chunk*. É possível que sejam solicitadas ao *Master* diversas solicitações de um mesmo cliente, dessa forma a resposta dessas requisições podem conter várias informações de diversos *chunks*. Esse conjunto de informações enviadas ao cliente auxilia no desempenho da aplicação, pois evita ao máximo a comunicação do *master* com o cliente.

O Google File System mostrou ser um modelo bastante eficiente para armazenamento de grandes conjuntos de dados. Além de garantir o armazenamento de grandes quantidades de dados, possui um alto poder de respostas às requisições dos clientes. Realizando um balanceamento de carga eficiente através de replicações de dados entre os servidores e utilizando estratégias para diminuir a quantidade de requisições ao *master*. Atualmente, esse sistema vem sendo utilizado constantemente pela Google em plataformas de armazenamento e

desenvolvimento do sistema, tanto para pesquisas quanto para serviços disponibilizados na *web*.

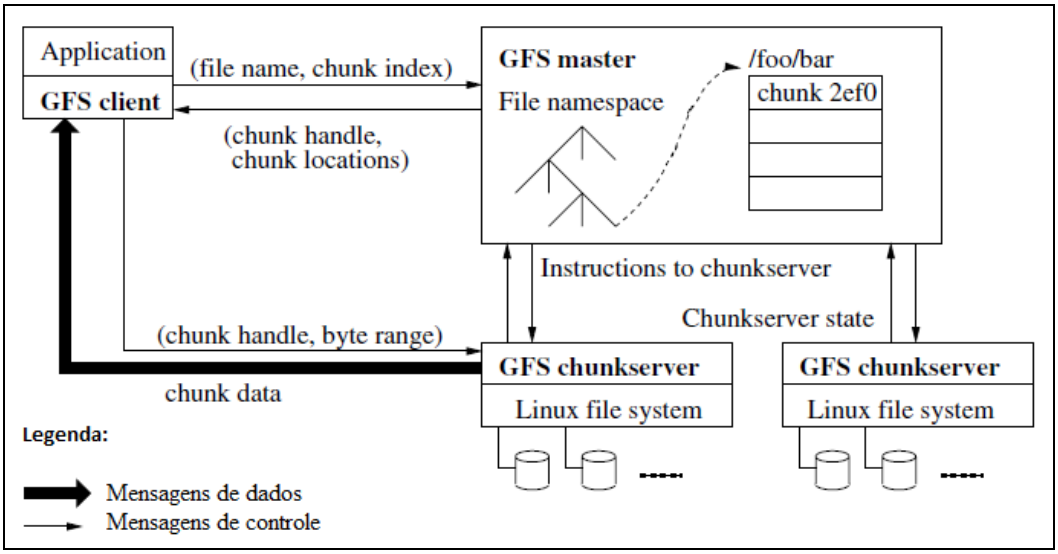

Figura 4.1: Modelo do Google File System. (GHEMAWAT; LEUNG; GOBIOFF, 2003, p. 3)

4.2 Hadoop

O Hadoop é uma versão *Open-Source* do *MapReduce*, desenvolvida na linguagem de programação Java e comumente utilizada para o processamento de grandes conjuntos de dados. Inicialmente foi criado por Doug Cutting e adquirido pela Apache Software Foudations, possui a Yahoo como uma das maiores organizações que investem neste projeto. Atualmente a popularização dessa ferramenta vem crescendo de forma acelerada. Empresas como o Facebook, IBM e Twitter adotaram esse sistema para o gerenciamento e manipulação das suas informações. Esta seção apresentará uma visão superficial do Hadoop, com ênfase nas técnicas abordadas pelo *MapReduce* e pelo processamento distribuído, demonstrando qual a sua finalidade dentro do universo da tecnologia da informação.

O grande sucesso da popularização do Hadoop entre diversas empresas é pelo fato de se tratar de um sistema capaz processar várias informações de aplicações em ambientes distribuídos, processando com eficiência e velocidade, grandes quantidades de dados. Trata-se de uma ferramenta executada em sistemas de *clusters*, para que clientes possam enviar requisições e receber respostas com alto desempenho, seguindo a linha de toda aplicação *MapReduce*: "*Write once, read many*". A grande vantagem da utilização deste sistema é a sua capacidade de ser escalável, robusto, acessível e fácil de implementar. É possível criar

códigos que possam ser processados em paralelo de forma rápida e simples, dentro de um ambiente com uma arquitetura linearmente escalável e tolerante a falhas. A Figura 4.2 apresenta uma ilustração de um sistema executando a ferramenta Hadoop.

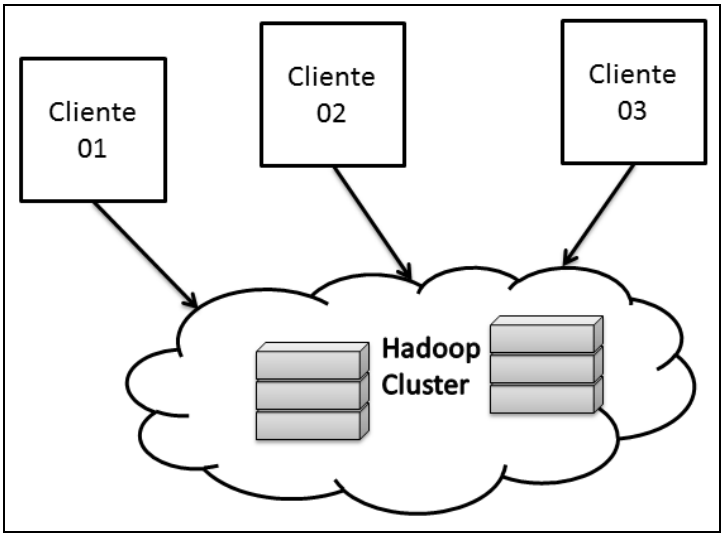

Figura 4.2: Utilização do Hadoop em um Cluster que aloca e processa grandes conjuntos de dados. O cliente solicita uma requisição e recebe uma resposta com alto desempenho.

Neste sistema cada solicitação do cliente é realizada dentro de uma nuvem e retornada rapidamente. A grande vantagem das aplicações Hadoop é o fato de utilizarem uma das características mais importantes do modelo *MapReduce*, a transferência de código entre os nós de um *cluster*. Dessa forma, não há necessidade de transmitir de dados por todo o ambiente. Apenas códigos são encaminhados para os nós que possuem uma determinada informação e então executados localmente.

Em um ambiente Hadoop, os nós são separados de forma hierárquica. Para o nó *master*, foi designado a nomenclatura de *jobTracker*, enquanto os *Workers* foram denominados *taskTrackers*, seguindo o mesmo raciocínio da organização vista no capitulo anterior. As entradas são separadas em blocos de 64 MB (por padrão), podendo ser alterada conforme a escolha do programador. Trata-se de arquivos intermediários denominados *chunks*, esses deverão ser processados em paralelo pelos nós de um *cluster*. A saída de um nó *reducer*, deverá ser replicada em um sistema de arquivos distribuídos.

4.2.1 Hadoop Distributed File System

Para entender melhor o funcionamento do Hadoop, é preciso entender como funciona o seu sistema de arquivos, conhecido como Hadoop Distributed File System (HDFS). Foi desenvolvido para armazenar grandes conjuntos de dados e uma alta capacidade de *streaming* de dados. O HDFS separa uma grande quantidade de informação em pequenos blocos e armazena esses blocos no sistema. A grande vantagem de incluir um arquivo em partes ocorre devido fato de um arquivo poder ser grande o suficiente para não caber em um único nó. Outro fator importante é a simplificação de um sistema de arquivos distribuídos, onde é possível verificar a quantidade de blocos que pode ser incluído em cada nó (sem a necessidade de criar arquivos de metadados para delimitar permissões de onde esses arquivos podem ser armazenados). Dados também podem ser replicados ao longo de todo o sistema, garantindo um ambiente tolerante a falhas e de alta disponibilidade.

No sistema HDFS, existe uma estruturação semelhante à hierarquia dos nós, seguindo a mesma visão do framework *MapReduce* (*Google*) e do Hadoop. Um nó *master* é denominado *namenode*, enquanto os nós *workers* são conhecidos como *datanodes*. O gerenciamento de todo o sistema de arquivos fica por parte do *namenode*. Ele mantém informações de todos os *datanodes* e a localização de cada bloco, além de conter uma árvore de arquivos e seus respectivos metadados, que armazena informações sobre esses arquivos e seus diretórios (WHITE, 2009, p. 65).

A grande dificuldade do HDFS é manter um sistema seguro e tolerante a falhas, porém toda essa preocupação só é direcionada aos *datanodes*. Caso o *namenode* venha a falhar, todas as informações quanto aos blocos de arquivos seriam perdidas. É de extrema importância garantir a vitalidade desse nó, pois caso haja alguma falha, todas as informações estariam perdidas e desencontradas, causando danos irreparáveis ao sistema. Para este tipo de situação, são realizadas rotinas de *backup* dos metadados e dos estados de cada *datanode*. Outra implementação do Hadoop para manter a segurança do *namenode*, é a criação de um nó auxiliar, cuja finalidade é garantir uma cópia, tal como uma imagem, do nó *master*.

A Figura 4.3 apresenta uma arquitetura do funcionamento do HDFS. O *namenode* gerencia e manipula todas as informações dos arquivos, tal como a localização e o acesso. Enquanto os *datanodes* se encarregam da leitura e escrita das informações nos sistemas de arquivos cliente.

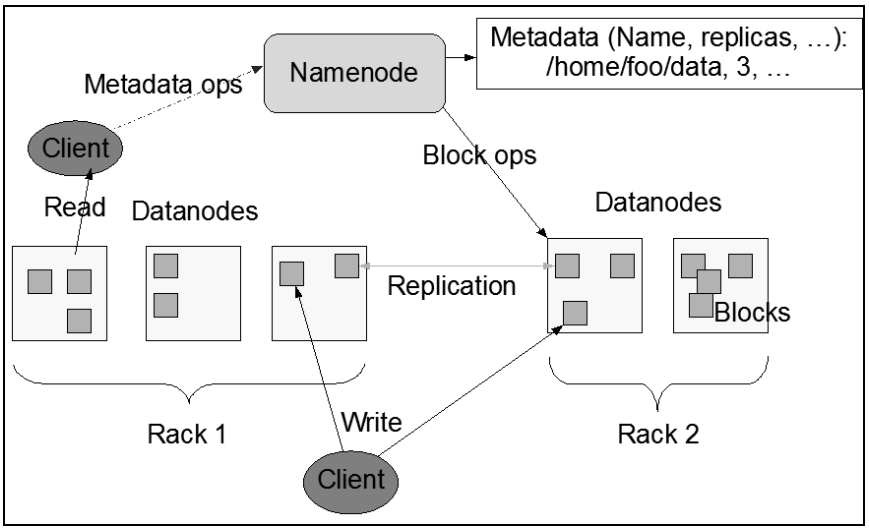

Figura 4.3: Arquitetura do modelo de arquivos do Hadoop (HDFS). (Fonte: http://hadoop.apache.org/common/docs/r0.17.0/hdfs_design.html).

Analisando a estrutura do HDFS e do Hadoop, é notável que ambos implementam o modelo *MapReduce*, e portanto é de suma importância esclarecer diferenças quanto às partes que compõem os dois sistemas. O Hadoop é um framework utilizado para aplicações distribuídas, enquanto o HDFS é utilizado para armazenamento das informações processadas pelo Hadoop. Em ambos sistemas é implementado o modelo *master-workers* proveniente do *MapReduce*.

4.2.2 Interface Hadoop

A grande dificuldade para um programador é verificar se seus dados estão consistentes dentro da aplicação. Uma vez que o modelo *MapReduce* trata todo e qualquer arquivo de um sistema em blocos distribuídos por diversos nós. Para este tipo de problema, a Apache utilizou *softwares* para capazes de verificar os dados contidos no Hadoop ou em seu sistema de arquivos. Esses tipos de programas são designados para fornecer uma *interface* ao sistema e são comumente utilizados no HDFS, por questões obvias, pois é necessário verificar se os dados estão sendo armazenados corretamente.

Uma das *interfaces* mais utilizadas por esse sistema é chamada de Thrift (WHITE, 2009, p. 49). Trata-se de uma *interface* que auxilia outras linguagens no acesso às informações dentro do sistema. Como mencionado no inicio deste capitulo, o Hadoop, assim como o HDFS, são sistemas desenvolvidos inicialmente na linguagem Java e logo possuem certas restrições quanto às aplicações que não foram desenvolvidas nessa linguagem. A

utilização do Thrift só possível com a utilização de um servidor Java em execução, pois ele fornece suporte à várias linguagens, tais como PHP, C++ e Ruby, através de *stubs* pré-gerados.

Outras *interfaces* são bem populares em sistemas HDFS e Hadoop como o FUSE, que utiliza uma biblioteca conhecida como libhdfs, desenvolvida na linguagem C para acessar os sistemas de arquivos do Hadoop. Essa biblioteca utiliza uma *interface* Java Nativa (JNI) para acessar as informações do sistema.

4.3 Aplicações MapReduce em Sistemas Reais

O *MapReduce* trouxe grandes benefícios a diversas aplicações. Um dos casos da utilização deste modelo em sistemas reais é a implantação da ferramenta Hadoop no *site* Last.fm[25] (WHITE, 2009, p. 405). Trata-se de uma aplicação voltada ao conteúdo musical, que ao longo do tempo evoluiu de milhares para milhões de pessoas. Os usuários dispõem de informações rápidas e precisas, manipulando, alocando e gerenciando os dados no sistema. Devido a essa sobrecarga, foi adotada a ferramenta Hadoop, que trouxe um bom resultado ao *site* em termos de desempenho e segurança.

O sistema Last.fm possui uma infra-estrutura de *clusters*, garantindo a escalabilidade da aplicação. Esse tipo de estrutura fornece uma melhor eficiência ao *site*. Outro beneficio trazido pelo Hadoop, é a utilização de *logs* de usuários e a replicação de dados dentro do sistema, fornecendo a segurança das informações de uma maneira simples e transparente. Por se tratar de um *software Open-Source*, é possível desenvolver novas funcionalidades ao sistema utilizando uma API clara e concisa.

Em 2009, o *site* Last.fm era composto de dois *clusters* Hadoop, com mais de 50 máquinas em processamento, utilizando 300 *cores* e com uma capacidade de 100 *terabytes*. Inúmeras operações são realizadas diariamente, gerando informações relevantes para o estudo da *performance* do sistema, como a análise de arquivos de *log*, avaliação de testes em máquinas e geração de gráficos do sistema.

Outro caso de sucesso do modelo *MapReduce*, é encontrado na rede social do jovem Mark Zuckerberg. O Facebook necessitava de um sistema que pudesse armazenar a sua

[25] O *site* Lasft.fm foi desenvolvido em 2002, com o intuito de oferecer serviços de radio e musica para os usuários. Calcula-se que 25 milhões de pessoas utilizam este *site* mensalmente. Disponível em <http://www.lastfm.com>. Acesso em 23 Mai.

imensa carga de *logs*, gerada diariamente pelos seus usuários. A base de dados utilizada não era suficiente para lidar com uma alta carga de informações. Para solucionar esse problema, foram colocadas algumas instâncias do Hadoop no sistema. O resultado foi bem sucedido, devido à simplicidade com que o sistema realizava suas operações. A evolução da ferramenta *Open-Source* de alta escalabilidade dentro do Facebook só se popularizou dentro da corporação, após o desenvolvimento do *Hive*[26], uma *interface* capaz de realizar consultas SQL em cima da plataforma Hadoop.

No ano de 2009, o Facebook possuía o segundo maior *cluster* Hadoop do mundo (WHITE, 2009, p. 415), com um espaço superior a 2PB, processando mais de 10TB de informações diariamente. O sistema possui um conjunto de 2.400 *cores* e cerca de 9TB de memória. O desenvolvimento da ferramenta Hive trouxe grandes benefícios ao Hadoop, sendo uma ferramenta adotada nos diversos subprojetos da Apache, que auxiliam no gerenciamento dos sistemas de alta escalabilidade.

[26] http://hive.apache.org/

5 CONCLUSÃO

A computação distribuída consiste em um conjunto de dispositivos trabalhando em paralelo, para atender com eficiência os serviços disponibilizados pelo sistema, seja ele um *cluster* ou um *grid*. A criação de um ambiente distribuído contribuiu com o desenvolvimento dos *datacenters*, que puderam aperfeiçoar seus recursos, sem a utilização de máquinas de grande porte, como no caso de um supercomputador. Desta forma, uma corporação poderia implantar um ambiente altamente robusto e eficiente, utilizando máquinas de pequeno porte e de baixo custo, se comparadas a uma máquina *high-end*.

A criação de um ambiente distribuído irá fornecer um sistema eficiente e com alto poder de resposta. Além de garantir maior disponibilidade e interoperabilidade dos serviços, utilizando uma infra-estrutura escalável e tolerante a falhas. Esse modelo organizacional trouxe grandes benefícios para aplicações de grande porte. Porém, trata-se de sistemas complexos, que precisam ser bem estruturados e para isso necessitam de profissionais que possuam grande conhecimento na área. Uma simples falha estrutural poderia causar danos cruciais para uma corporação. Outra desvantagem está relacionada ao desenvolvimento de aplicações em ambientes dessa natureza. É necessário que o programador tenha um alto conhecimento, para que possa desenvolver serviços em aplicações distribuídas. Para esse problema, foram criados mecanismos capazes de abstrair o problema da implementação ambientes distribuídos, utilizando *interfaces* e padrões, tais como a OpenMP e a MPI.

O *MapReduce* mostrou ser um excelente modelo para a computação distribuída. É possível criar aplicações que possam processar *terabytes* de informações sem causar uma exagerada sobrecarga ao sistema. Dessa forma, milhares de usuários podem desfrutar de sistemas que contenham inúmeras informações, com um elevado poder de resposta. A abordagem deste modelo também se baseia no conceito de transparência, auxiliando no desenvolvimento de aplicações, sem que o programador tenha experiência alguma com ambientes distribuídos. Trata-se de um sistema seguro, que possui um mecanismo de processamento paralelo, utilizando técnicas de balanceamento de carga e um alto poder de indexação de dados.

O modelo possui simplicidade em sua arquitetura, onde os dispositivos são separados conforme suas funções. Cada informação é processada localmente, ao invés de ser transmitida pela rede, evitando um aumento no tráfego de dados dentro do sistema. Outra preocupação que o modelo possui, é a probabilidade da ocorrência de problemas entre os nós do sistema.

Nesse caso, a importância de desenvolver um sistema tolerante a falhas é garantida a partir de vários mecanismos, como o uso da replicação de dados e a verificação de *logs* de cada dispositivo, analisando o estado e o comportamento de cada componente.

É importante afirmar que a utilização da computação distribuída garantiu uma maior expansão dos sistemas. É possível encontrar milhões de usuários comunicando-se e conectando-se graças ao poder dessas aplicações. Atualmente a quantidade de informações geradas pelo universo da *web*, garante que possamos conhecer ambientes e realizar trocas de informações com o menor esforço possível. É assim que podemos imaginar o futuro para a computação, um ambiente altamente escalável com uma alta capacidade de armazenamento e um elevado poder de resposta, com o menor custo possível.

O *MapReduce* é apenas o um dos modelos empregados dentro do âmbito da computação distribuída. Novas pesquisas são realizadas todos os anos, e novos algoritmos estão surgindo, com um maior poder de processamento, como no caso do novo sistema denominado *Caffeine*[27], utilizado para realizar buscas mais rápidas e melhores. Segundo o Google, idealizador do novo sistema, é possível realizar buscas de *sites* com até 50% a mais de velocidade do que o motor de busca atual, proveniente do *MapReduce*.

[27] http://googleblog.blogspot.com/2010/06/our-new-search-index-caffeine.html

REFERÊNCIAS BIBLIOGRÁFICAS

MEIRA, Silvio Romero de Lemos. **Introdução a programação funcional**. Campinas, SP: UNICAMP. 1988. 290p.

DE SÁ, Claudio Cesar e DA SILVA, Marcio Ferreira. **Haskell: Uma abordagem Prática.** Novatec. 2006. 296p.

PETRICEK, Tomas e SKEET, Jon. **Real World Functional Programming.** Manning Pubns Co. 2009. 529p.

THOMPSON, Simon. **Haskell: The Craft of Functional Programming.** 2.ed. Addison-Wesley. 1999. 528p.

HINDLEY, J. Roger e SELDIN, Jonathan P. **Lambda-Calculus and Combinators, an Introduction.** Cambridge University Press. 2008. 345p.

WITTEN, Ian H. e FRANK, Eibe. **Data Mining: Pratical Machine Learning Tools and Techniques**. Elsevier. San Francisco, CA. 2005. 525p.

KSHEMKALYANI, Ajay D. e SINGHAL, Mukesh. **Distributed Computing: Principles, Algorithms, and Systems**. Cambridge. Cambridge University. 2008. 736p.

LIN, Jimmy and DYER, Chris. **Data-Intensive Text Processing with MapReduce**. Maryland, 2010. Final Pre-Production. Disponível em <http://www.umiacs.umd.edu/~jimmylin/MapReduce-book-final.pdf>. Acesso 10 Jun.

DEAN, Jeffrey. and GHEMAWAT, Sanjay. **MapReduce: Simplified Data Processing on Large Clusters**. In: Proceedings of Sixth Symposium on Operating System Design and Implementation, San Francisco, CA. December, 2004. Disponível em <http://labs.google.com/papers/mapreduce-osdi04.pdf>. Acesso em 10 Jun.

GHEMAWAT, Sanjay. et al. **Bigtable: A Distributed Storage System for Structured Data**. In: Proceedings of Seventh Symposium on Operating System Design and Implementation, Seattle, WA. November, 2006. Disponível em <http://labs.google.com/papers/bigtable-osdi06.pdf>. Acesso em 10 Jun.

GHEMAWAT, Sanjay, LEUNG, Shun-Tak and GOBIOFF, Howard. **The Google File System**. In: 19th ACM Symposium on Operating Systems Principles, Lake George, NY. October, 2003. Disponível em <http://labs.google.com/papers/gfs-sosp2003.pdf>. Acesso em 10 Jun.

MOORE, Gordon E. **Cramming more components onto integrated circuit.** Electronics Magazine. Ano. 8. N. 38, abr. 1965. Disponível em <ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf>. Acesso em 10 Jun.

CORMEN, Thomas H., LEISERSON, Charles E., RIVEST, Ronald L. and STEIN, Clifford. **Introduction to Algorithms.** 2 ed. McGraw-Hill Book Co. 2002. 1180p.

GRAMA, Ananth, GUPTA, Anshul, KARYPIS, George e KUMAR, Vipin. **Introduction to Parallel Computing.** 2 ed. Addison Wesley. 2003. 856p.

PAGINA PESSOAL Karatsuba AnatoliiAlexeevich. Russian Academy of Science: Steklov Mathematical Institute, 2011. Apresenta informações sobre o autor que desenvolveu o algoritmo de Karatsuba. Disponível em <http://www.mi.ras.ru/~karatsuba/index_e.html>. Acesso em 10 Jun.

ALEXEEVICH, Karatsuba A. **The Complexity of Computations.** In: Proceedings of Steklov Institute of Mathematics. Vol 211. p. 169-183. Moscou. Janeiro, 1995. Disponível em <http://www.ccas.ru/personal/karatsuba/divcen.pdf>. Acesso em 10 Jun.

LAM, Chuck. **Hadoop in Action.** Manning Publications Co. 2011. 311p.

WHITE, Tom. **Hadoop: The Definitive guide**. O'Reilly. 2009. 501p

RAJARAMAN, Anand e ULLMAN, Jeffrey D. **Mining of Massive Datasets**. Stanford University. 2010. 320p. Disponível em <http://infolab.stanford.edu/~ullman/mmds.html>. Acesso em 10 Jun.

SILBERSCHATZ, Abraham, GALVIN, Peter P. e GAGNE, Greg. **Sistemas Operacionais: Conceitos e Aplicações.** Editora Campus. 1999. 571p.

SILBERSCHATZ, Abraham, GALVIN, Peter P. e GAGNE, Greg. **Fundamentos de Sistemas Operacionais.** 6 ed. LTC Editora. 2002. 557p.

CARNEGIE MELLON UNIVERSITY. **The Andrew Project: What is Andrew?** Disponível em <http://www.cmu.edu/corporate/news/2007/features/andrew/what_is_andrew.shtml>. Acesso em 10 Jun.

ANTONOPOULOS, Nick e GILLAM, Lee. **Cloud Computing: Principles, Systems and Applications**. Springer. 2010. 373p.

VELTE, Anthony T., VELTE, Toby J. e ELSENPETER, Robert. **Cloud Computing: A Practical Approach**. McGrall Hill Co. 2010. 327p.

APACHE HADOOP. **The Hadoop Distributed File System: Architecture and Design.** Disponível em <http://hadoop.apache.org/common/docs/r0.17.0/hdfs_design.html>. Acesso em 10 Jun.