JOÃO EUGENIO MARYNOWSKI

UMA ABORDAGEM PARA O TESTE DE DEPENDABILIDADE DE SISTEMAS MAPREDUCE COM BASE EM CASOS DE FALHA REPRESENTATIVOS

Tese apresentada como requisito parcial à obtenção do título de Doutor em Ciência da Computação, no Programa de Pós-Graduação em Informática do Setor de Ciências Exatas da Universidade Federal do Paraná.
Orientador: Prof. Andrey Ricardo Pimentel

CURITIBA
2013

M393a Marynowski, João Eugenio
Uma abordagem para o teste de dependabilidade de sistemas MapReduce com base em casos de falha representativos / João Eugenio Marynowski. – Curitiba, 2013.
100f. : il. color. ; 30 cm.

Tese (doutorado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática, 2013.

Orientador: Andrey Ricardo Pimentel.
Bibliografia: p. 94-100.

1. Processamento eletronico de dados -- Processamento distribuido. 2. Software - Verificação. 3. Tolerância a falhas (Computação). I. Universidade Federal do Paraná. II. Pimentel, Andrey Ricardo del. III. Título.

CDD: 004.36

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa do aluno de Doutorado em Ciência da Computação, João Eugenio Marynowski, avaliamos a tese de doutorado intitulada *"Uma Abordagem para o Teste de Dependabilidade de Sistemas MapReduce com base em Casos de Falha Representativos"*, cuja defesa pública foi realizada no dia 08 de novembro de 2013, às 09:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após avaliação, decidimos pela: (X) **aprovação** do candidato. ( ) **reprovação** do candidato.

Curitiba, 08 de novembro de 2013.

Prof. Dr. Andrey Ricardo Pimentel
DINF/UFPR – Orientador

Profa. Dra. Taisy Silva Weber
UFRGS – Membro Externo

Prof. Dr. Marco Vieira
Univ. de Coimbra – Membro Externo

Prof. Dr. Adilson Luiz Bonifácio
UEL – Membro Externo

Prof. Dr. Ricardo Lüders
UTFPR – Membro Externo

Prof. Dr. Elias Procópio Duarte Júnior
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

Dedico este trabalho a minha querida esposa, Corina, ao meu pai, Eugenio, minha mãe, Marília, e aos meus filhos Amanda, Pedro e Daniel.

AGRADECIMENTOS

Agradeço primeiramente a Deus por sua indiscutível importância; sem Ele nada é possível.

Agradeço a minha família, especialmente minha esposa Corina e meu filhos Amanda, Pedro e Daniel, por todo carinho, amor e ajuda incondicionais; aos meus pais, Eugenio e Marília, que me mostraram por onde caminhar e me ajudaram sempre que precisei; ao meu sogro Roberto e minha sogra Adelaide; minhas tias Sirlene, Salete e Tonha; e todos os outros membros dessa família que eu tanto amo.

Aos professores, Andrey Pimentel e Eduardo Almeida por terem me orientado durante este trabalho; à Taisy Weber, Marco Vieira, Adilson Bonifácio, Ricardo Lüders, Gerson Sunye, Bona e Elias pelas contribuições e sugestões; ao Fabiano, Letícia, André Guedes, Sunye, Castilho, Direne, Didonet, e outros, que me auxiliaram sempre que precisei.

Aos funcionários, Jucélia, Rafael, Letícia, Raquel, Otávio, Armando, Osvaldo, Andrea e Michele, e as zeladoras Sônia, Marilda, Zenaide, Zenilda, Senhorinha, ente outras que me propiciaram um excelente ambiente de estudo e trabalho.

Aos colegas de percurso que me ajudaram e propiciaram vários momentos de descontração, incluindo o Michel, Ramiro, Tarcízio, Rebeca, Razer, Eduardo Sant'Ana, Ailton, Ricardo, Cristiane (em memória), Guilherme, Luiz, Jorge, Arion, Ribas, Willian, Alexander e em especial Roverli e Antonio que me ajudaram nas revisões deste trabalho e dos meus artigos.

Por fim, agradeço também a todos meus amigos que estão sempre próximos e rezam por mim, incluindo o André, Juliana, João, Ana, Luciano, Suelen, Anderson, Rosélis, Emerson, Adriana, Gabriel, Flávia, Egon, Thiago, Renata, Pedro, Renata, Zeno, Andréia, entre outros que de alguma forma contribuíram para a realização deste trabalho.

Muito obrigado!

"Contudo, seja qual for o grau a que chegamos, o que importa é prosseguir decididamente."

Filipenses 3, 16.

RESUMO

Os sistemas MapReduce facilitam a utilização de um grande número de máquinas para processar uma grande quantidade de dados, e têm sido utilizados por diversas aplicações, que incluem desde ferramentas de pesquisa até sistemas comerciais e financeiros. Uma das principais características dos sistemas MapReduce é abstrair problemas relacionados ao ambiente distribuído, tais como a distribuição do processamento e a tolerância a falhas. Com isso, torna-se imprescindível garantir a dependabilidade dos sistemas MapReduce, ou seja, garantir que esses sistemas funcionem corretamente mesmo na presença de falhas. Por outro lado, a falta de determinismo de um ambiente distribuído e a falta de confiabilidade do ambiente físico, podem gerar erros nos sistemas MapReduce que sejam difíceis de serem encontrados, entendidos e corrigidos. Esta tese apresenta a primeira abordagem conhecida para o teste de dependabilidade para sistemas MapReduce. Este trabalho apresenta uma definição para o teste de dependabilidade, uma modelagem do mecanismo de tolerância a falhas do MapReduce, um processo para gerar casos de falha representativos a partir de um modelo, e uma plataforma de teste para automatizar a execução de casos de falha em um ambiente distribuído. Este trabalho ainda apresenta uma nova abordagem para modelar componentes distribuídos usando redes de Petri. Essa nova abordagem permite representar a dinâmica dos componentes e a independência de suas ações e estados. Resultados experimentais são apresentados e mostram que os casos de falha gerados a partir do modelo são representativos para o teste do sistema Hadoop, principal implementação de código aberto do MapReduce. Através dos experimentos, diversos erros são encontrados no Hadoop, e os resultados também comprovam que a plataforma de teste automatiza a execução dos casos de falha representativos. Além disso, a plataforma apresenta as propriedades requeridas para uma plataforma de teste, que são a controlabilidade, medição temporal, não-intrusividade, repetibilidade, e a eficácia na identificação de sistemas com erros.

Palavras-chave: Teste. Validação. Dependabilidade. Tolerância a Falhas.

ABSTRACT

MapReduce systems allow the use of a large number of machines to process a big amount of data, and have been used for several applications, ranging from search engines to financial and commercial systems. A key feature of MapReduce systems is abstract distributed environment issues, such as fault tolerance and processing distribution. Thus, it is essential to ensure the dependability of MapReduce systems, i.e., ensure that these systems can execute correctly and without interruption, even in the presence of failures. On the other hand, the lack of determinism in a distributed environment and the lack of reliability of physical environments may cause errors in MapReduce systems that are difficult to find, to define and to correct. This thesis presents the first known approach to the dependability test of MapReduce systems. This work presents a dependability testing definition, a MapReduce fault tolerance mechanism model, a process to generate representative fault cases from the model, and a testing framework to automate the execution of the fault cases in a distributed environment. This work also presents a new approach to model distributed components using Petri nets. The new approach allows representing the dynamic of the components and the independence of their actions and states. Experimental results are presented and show that the generated fault cases are representative to test the Hadoop system, that is the main MapReduce open source implementation. Several errors are identified and the experiments also confirmed that the testing framework automates the execution of representative fault cases. Furthermore, the presented framework includes the required properties for a testing platform, that are: controllability, time measurement, nonintrusiveness, repeatability, and the effectiveness in the identification of system errors.

Key-words: Test. Validation. Dependability. Fault Tolerance.

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

A quantidade de dados gerada e armazenada por diversas aplicações, tais como redes sociais, aplicações comerciais e de pesquisa, já passa da casa de *petabytes* e tende a aumentar drasticamente com o passar dos anos [Winter and Kostamaa, 2010, Zaharia et al., 2008]. Diversos sistemas computacionais facilitam o armazenamento e o processamento dessa grande quantidade de dados, que por sua vez, é distribuída em um grande número de máquinas [Ghemawat et al., 2003, Chang et al., 2008]. Exemplos desses sistemas incluem o Scope [Chaiken et al., 2008], o Clustera [DeWitt et al., 2008], Sector and Sphere [Gu and Grossman, 2009], e em especial o MapReduce [Dean and Ghemawat, 2004] – o sistema mais popular entre eles e que é utilizado por grandes corporações, como Google, Ebay, Yahoo, Facebook, LinkedIn, e Quantcast [Apache, 2013d].

Um sistema MapReduce possibilita que seus usuários utilizem facilmente milhares de máquinas sem se preocupar com os problemas de um ambiente distribuído, tais como o particionamento de dados, a distribuição de processamento, a replicação e a tolerância a falhas. O usuário escreve apenas uma aplicação composta por duas funções de alto nível, denominadas *Map* e *Reduce*, e então submete a aplicação ao sistema para que seja distribuída e executada. A função *Map* mapeia o dado de entrada que está distribuído nas máquinas, enquanto a função *Reduce* agrupa o resultado da função *Map*, e gera o resultado final do processamento.

Assim como qualquer outro sistema distribuído que utiliza milhares de máquinas, um sistema MapReduce frequentemente enfrenta falhas devido a diversas condições, como interrupções elétricas, problemas em *hardware*, e atualizações ou defeitos em *softwares* [Liu et al., 2004, Henry, 2009, Schroeder and Gibson, 2006]. A tolerância a falhas faz-se ainda mais importante ao se considerar o número crescente de empresas comerciais e financeiras que pretendem utilizar os sistemas MapReduce para processar as suas grandes quantidades de dados, tais como

Bank of America, MorganStanley, GE e InfoChimps [Apache, 2013d, Forbes, 2013, Eventbrite, 2013]. Com isso, torna-se imprescindível garantir que as falhas não interfiram ou interrompam a execução de um sistema MapReduce.

A dependabilidade (*dependability*) é a propriedade de um sistema que define a sua habilidade em evitar interferências e interrupções no seu funcionamento, mesmo na presença de falhas [Avizienis et al., 2004]. A dependabilidade é um conceito integrador que compreende a disponibilidade (*availability*), confiabilidade (*reliability*), segurança (*safety*), integridade (*integrity*), e a manutenabilidade (*maintainability*) de um sistema. A dependabilidade dos sistemas MapReduce pode ser garantida através da realização do seu teste. O teste é a validação de um sistema pela observação de sua execução, e tem como objetivo encontrar comportamentos diferentes do especificado, isto é, a presença de erros [Ammann and Offutt, 2008].

A *injeção de falhas* é uma técnica utilizada para a avaliação da dependabilidade de sistemas distribuídos. Ela consiste na injeção artificial de falhas enquanto o sistema é executado em um ambiente controlado, e tem com objetivo avaliar o comportamento do sistema [Arlat et al., 2003, Hsueh et al., 1997, Voas, 1997]. Um *caso de falha* é a combinação de todas as informações necessárias para uma completa execução e validação do sistema em teste, e envolve a injeção de falhas [Echtle and Leu, 1994]. A *representatividade de um caso de falha* diz respeito ao seu impacto na validação de um sistema, ou seja, qual a importância do caso de falha na identificação de erros desse sistema [Arlat et al., 2003, Natella et al., 2012].

Os trabalhos relacionados com a dependabilidade de sistemas MapReduce avaliam o comportamento desses sistemas mediante falhas, mas são insuficientes para validar o sistema considerando a sua dependabilidade. Portanto, a tese a ser defendida neste trabalho é expressada na seguinte questão:

> "É possível e vantajoso criar uma abordagem para o teste de dependabilidade de sistemas MapReduce com base na geração e execução de casos de falha representativos?"

1.1 Objetivos

O objetivo geral deste trabalho é responder essa pergunta apresentando uma abordagem para a realização do teste de dependabilidade para sistemas MapReduce. Essa abordagem é baseada na execução automática de casos de falha representativos gerados a partir do modelo formal do mecanismo de tolerância a falhas do MapReduce.

Os objetivos específicos dessa tese são: (1) modelar o mecanismo de tolerância a falhas do MapReduce; (2) gerar os casos de falha representativos a partir desse modelo, e (3) automatizar a execução de casos de falha em um ambiente distribuído.

Para modelar o mecanismo de tolerância a falhas de um sistema MapReduce é necessário o uso de um modelo formal capaz de representar a dinâmica de um sistema MapReduce. Dois principais modelos formais aplicados para sistemas distribuídos serão analisados: a máquina de estados finita [Ural, 1992] e a rede de Petri [Murata, 1989].

Um processo para gerar os casos de falha a partir do modelo formal será apresentado. Os casos de falha obtidos são considerados representativos e devem conter as informações necessárias para que sejam executados manualmente ou automaticamente a partir de uma plataforma de teste.

A automatização da execução dos casos de falha será possibilitada a partir de uma plataforma de teste denominada HadoopTest. Essa plataforma será apresentada em detalhes, principalmente informando como ela controla e monitora individualmente cada componente distribuído do MapReduce, e como ela injeta falhas de acordo com as etapas de processamento dos componentes.

1.2 Justificativa

Para realizar o teste de dependabilidade de um sistema MapReduce é necessário resolver dois problemas principais. O primeiro é a geração de um conjunto de casos de falha que sejam representativos para a realização do teste de dependabilidade. Essa tarefa envolve a escolha de elementos de um conjunto infinito e parcialmente desconhecido de casos de falha que podem

ocorrer em um ambiente distribuído [Voas, 1997]. O segundo problema é a automatização da execução dos casos de falha através de uma plataforma de teste. Por sua vez, esta atividade envolve o controle individualizado dos componentes do MapReduce distribuídos em diversas máquinas.

O mecanismo de tolerância a falhas do MapReduce trata as falhas de forma diferenciada dependendo do momento em que ocorrem. Por exemplo, o tratamento de uma falha que ocorre em uma máquina executando uma tarefa de *Map* é diferente de uma falha que ocorre após a finalização dessa tarefa, que por sua vez é diferente de uma falha que ocorre enquanto a máquina executa uma tarefa de *Reduce*. Os casos de falha representativos para o teste de dependabilidade de um sistema MapReduce devem ser gerados a partir de uma especificação ou modelo formal do mecanismo de tolerância a falhas.

Os casos de falha utilizados nos trabalhos que abordam a dependabilidade dos sistemas MapReduce são definidos pelo engenheiro de teste e dizem respeito à avaliação do sistema e não à sua validação [Dean and Ghemawat, 2004, Abouzeid et al., 2009, Sangroya et al., 2012a]. Os casos de falha gerados ignoram os diferentes comportamentos do mecanismo de tolerância a falhas, injetando falhas sem considerar as tarefas e as etapas de processamento. Por exemplo, os casos de falha utilizados envolvem a injeção de falhas em algumas máquinas (falhando 3 de 10) por um período (10 segundos, iniciando 30 segundos a partir do início da execução). Embora essas abordagens possibilitem avaliar alguns atributos da dependabilidade dos sistemas MapReduce, elas são insuficientes para o teste de dependabilidade, uma vez que não validam o comportamento do sistema injetando falhas em todas as suas etapas de processamento.

Algumas plataformas existentes e que realizam o teste de sistemas distribuídos de grande escala, tais como o MapReduce e P2P [Androutsellis-Theotokis and Spinellis, 2004], apresentam soluções para o controle dos componentes distribuídos e para a validação do seu comportamento [Pan et al., 2010, Butnaru et al., 2007, Zhou et al., 2006, de Almeida et al., 2010b]. Por outro lado, essas plataformas não consideram a injeção de falhas e nem controlam o momento específico do sistema em que as ações devem ser executadas.

Outras plataformas empregadas para a injeção de falhas em sistemas distribuídos suportam a execução de casos de falha com a injeção de diversas e múltiplas falhas, mas não controlam a execução do sistema [Jacques-Silva et al., 2006, Pham et al., 2011, Stott et al., 2000, Lefever et al., 2004, Hoarau et al., 2007]. Essa característica impossibilita identificar dinamicamente em quais componentes e em quais etapas de processamento as falhas devem ser injetadas.

O Hadoop foi a implementação do MapReduce escolhida como estudo de caso para a realização do teste de dependabilidade apresentado nesta tese. Resultados experimentais são apresentados e mostram que os casos de falha gerados a partir do modelo são representativos para o teste do sistema Hadoop. Através dos experimentos, diversos erros foram encontrados, e os resultados também mostram que a plataforma de teste automatiza a execução dos casos de falha representativos. Além disso, a plataforma apresenta as propriedades requeridas para uma plataforma de teste, que são a controlabilidade, medição temporal, não-intrusividade, repetibilidade, e a eficácia na identificação de sistemas com erros.

1.3 Organização deste Trabalho

O restante deste trabalho está organizado da seguinte forma. O Capítulo 2 apresenta os fundamentos e os trabalhos relacionados ao teste de dependabilidade de sistemas MapReduce. O Capítulo 3 apresenta a análise para a modelagem do mecanismo de tolerância a falhas e o processo para a geração dos casos de falha representativos. Na sequência, o Capítulo 4 apresenta o HadoopTest, que é a plataforma de teste desenvolvida para a automatização da execução dos casos de falha representativos. O Capítulo 5 apresenta os resultados experimentais obtidos na utilização do HadoopTest para testar a dependabilidade do Hadoop aplicando os casos de falha representativos. O Capítulo 6 traz as conclusões e os trabalhos futuros, e por fim, segue um registro das publicações realizadas neste doutorado.

CAPÍTULO 2

ESTADO DA ARTE

Diversos sistemas computacionais facilitam o armazenamento e o processamento de uma grande quantidade de dados, distribuída em um grande número de máquinas. O MapReduce [Dean and Ghemawat, 2004] é um exemplo popular desses sistemas. O MapReduce possibilita a utilização de milhares de máquinas e tolera naturalmente a falha em algumas delas. Por outro lado, garantir a tolerância a falhas de um sistema não é uma tarefa trivial, pois envolve validar o comportamento do sistema mediante todas as falhas toleradas. O objetivo do teste de dependabilidade é encontrar erros no sistema considerando o seu mecanismo de tolerância a falhas, e assim garantir o seu funcionamento.

Este capítulo apresenta os principais conceitos e fundamentos utilizados no teste de dependabilidade para sistemas MapReduce. Inicialmente, o capítulo apresenta a terminologia para o teste de software, o conceito de dependabilidade, e a técnica de injeção de falhas. Na sequência, o MapReduce é apresentado em detalhes, juntamente com o seu mecanismo de tolerância a falhas, que são os requisitos necessários para a realização do teste de dependabilidade desse sistema. Por fim, os trabalhos relacionados com a geração de casos de falha representativos e a automatização da execução desses casos de falha são apresentados.

A maior parte da terminologia apresentada é baseada no livro "Introdução ao Teste de Software", escrito por Paul Ammann e Jeff Offutt [Ammann and Offutt, 2008]. As terminologias também seguem o padrão internacional para plataformas e metodologias de teste de conformidade (Conformance Testing Methodology and Framework - CTMF) definida na ISO/IEC 9646 [ISO/IEC, 1994].

2.1 Teste de Software

O teste de software é uma investigação conduzida para prover informações sobre a qualidade de um sistema em teste, ou SUT (*System Under Test*). O engenheiro de teste é o profissional responsável pela realização do teste. Por sua vez, o teste envolve a criação e execução de casos de teste, análise dos resultados, e relato aos responsáveis sobre os comportamentos obtidos. A seguir são apresentados os principais termos de teste de software que serão utilizados no restante deste trabalho.

Definição 2.1.1 (Defeito) *Um defeito (defect ou bug) é uma imperfeição estática no SUT, ou seja, uma implementação incorreta.*

Definição 2.1.2 (Erro) *Um erro (error) é um estado incorreto do SUT gerado a partir da manifestação de algum defeito.*

Definição 2.1.3 (Teste) *É a validação do SUT pela observação de sua execução com o objetivo de encontrar erros.*

Definição 2.1.4 (Caso de Teste) *É uma combinação dos componentes necessários para uma completa execução e validação do SUT, sendo eles: os valores de entrada e os valores esperados.*

Definição 2.1.5 (Valor de Entrada) *É o valor de entrada necessário para uma completa execução do SUT.*

Definição 2.1.6 (Valor Esperado) *É o valor que deve ser produzido pelo SUT ao final da execução do teste.*

O valor esperado e o resultado obtido a partir da execução de um caso de teste são usados para associar um *veredito* para o SUT na execução do caso de teste.

Definição 2.1.7 (Veredito) *É o resultado de um caso de teste, podendo ser: PASSOU, FALHOU, ou INCONCLUSIVO.*

Se o resultado do caso de teste for igual ao valor esperado, o veredito do SUT na execução do caso de teste é *PASSOU*, caso contrário, o veredito é *FALHOU*. O veredito é *INCONCLUSIVO* quando o resultado do caso de teste não é preciso o suficiente para aferir nenhuma das afirmações anteriores, e precisa ser refeito.

Definição 2.1.8 (Oráculo) *É o mecanismo responsável por associar um veredito, ou seja, verificar se o SUT comportou-se corretamente em uma execução de um caso de teste.*

Para um melhor entendimento das definições apresentadas até aqui, o Algoritmo 1 apresenta a função *ultimoZero*. Essa função objetiva retornar a posição do último elemento igual a zero de um vetor. A função *ultimoZero* recebe como parâmetro de entrada um vetor de inteiros V e retorna a posição do último elemento igual a zero (0) que existir em V, ou retorna o valor -1 caso não exista um elemento igual a zero.

Algoritmo 1: ultimoZero

```
Entrada: V, um vetor de valores inteiros.
Dados: i, um valor inteiro.
Saída: um valor inteiro.
Início
    i ← 0
    Enquanto i < |V| faça
        Se V[i] = 0 então
            Retorna i
        fim
        i ← i + 1
    fim
    Retorna −1
fim
```

Para testar o sistema representado por esse algoritmo, inicialmente foi considerado o seguinte caso de teste:

— Valores de Entrada: $V = [2, 7, 0]$

— Valor Esperado: 2

A execução desse caso de teste pelo sistema obtém como resultado o valor 2. Utilizando um oráculo baseado na comparação de valores tem-se que, o resultado obtido é igual ao

resultado esperado, portanto o veredito para esse caso de teste é *PASSOU*, ou seja, o sistema não apresentou erro na sua execução.

Agora, considere o seguinte caso de teste:

— Valores de Entrada: $V = [0, 7, 0]$

— Valor Esperado: 2

Esse caso de teste fornece como valor de entrada um vetor de inteiros que possui duas ocorrências do valor 0, uma na posição 0, e outra na posição 2. Portanto, o valor esperado é 2, já que segundo a definição do sistema, ele deve retornar a posição da última ocorrência de um valor 0 no vetor de entrada. Entretanto, o resultado da execução do caso de teste é o valor 0 (zero). Comparando esse valor com o valor 2 esperado, o veredito desse caso de teste é *FALHOU*. O veredito *FALHOU* indica que o sistema não comportou-se conforme especificado, ou seja, revela um *erro* no sistema. É importante ressaltar que ambos os casos de teste exercitam o defeito, embora apenas no segundo caso é que o erro ocorre.

Definição 2.1.9 (Depuração) *Processo de encontrar um defeito dado um erro de um sistema.*

Considere novamente o sistema que implementa a função *ultimoZero*, mostrado no Algoritmo 1. Depurando esse sistema, é possível identificar que o seu *defeito* está no fato da função *ultimoZero* retornar a posição da primeira ocorrência de um elemento igual a zero do vetor V. Este comportamento está inconsistente com o especificado, indicando que o método deve retornar a posição da última ocorrência de um elemento igual a zero do vetor.

Dois problemas práticos associados ao teste de sistemas estão relacionados à capacidade de fornecer os valores de entrada corretamente para o sistema, e de observar detalhes de seu comportamento. Estas duas definições são usadas para aperfeiçoar a definição de caso de teste.

Definição 2.1.10 (Observabilidade) *Problema de observar o comportamento do SUT em termos de seus componentes, resultados, e efeitos no ambiente.*

Definição 2.1.11 (Controlabilidade) *Problema de prover a um SUT suas entradas no momento adequado, em termos de valores e operações.*

O engenheiro de teste é o profissional responsável por definir e executar os casos de teste sobre o SUT, avaliando os resultados do processo de teste para garantir a qualidade do sistema. A Figura 2.1 ilustra o processo de teste de um sistema. Após o desenvolvimento do caso de teste, ele é executado pelo SUT que gera um resultado. O resultado e os valores esperados são utilizados no oráculo para associar um veredito. Se o veredito for *INCONCLUSIVO* é feita uma nova execução do caso de teste pelo SUT. Se o veredito for *FALHOU*, indica que o comportamento do SUT durante o teste foi diferente do especificado, e que o processo de teste acaba com a detecção de um *erro* no sistema. O processo de teste também é finalizado quando o veredito é *PASSOU* e foi alcançado um certo critério de parada, por exemplo, obter três execuções do caso de teste com o veredito *PASSOU*.

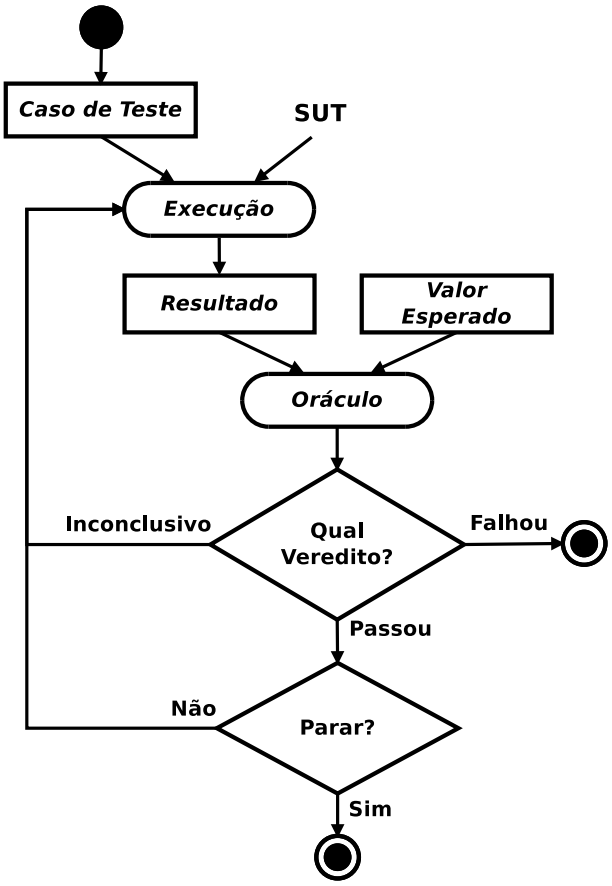


Figura 2.1: Processo de teste (adaptado de [de Almeida, 2009]).

A automatização do processo de teste é uma propriedade desejável sempre que possível. Isso pode ser feito por um sistema que possa controlar o SUT, executar o caso de teste, implementar um oráculo, e produzir um relatório claro sobre o teste.

Definição 2.1.12 (Plataforma de Teste) *É um sistema capaz de controlar, monitorar e avaliar o SUT mediante a execução de um caso de teste.*

A arquitetura clássica de uma plataforma de teste capaz testar um sistema distribuído é formada por um componente *coordenador* e vários componentes *testador* [ISO/IEC, 1994, Walter et al., 1998]. Cada testador controla a execução de um componente do SUT. O coordenador orquestra a execução dos testadores e gera o veredito do caso de teste.

Cada componente de um sistema distribuído pode ter um comportamento distinto. Esses comportamentos são identificados como ações de um caso de teste distribuído, conforme apresentado por Almeida et al. [de Almeida et al., 2008, de Almeida, 2009, de Almeida et al., 2010c]

Definição 2.1.13 (Caso de Teste Distribuído) *É um caso de teste que se aplica a um sistema distribuído e é formado por um conjunto de ações que devem ser executadas por um conjunto de testadores.*

A validação da execução de um caso de teste distribuído ocorre a partir do resultado da execução das ações por cada testador. Os resultados das ações são usadas pelo oráculo que atribui um veredito para um caso de teste distribuído.

A próxima seção deste capítulo apresenta o conceito de dependabilidade e a técnica de injeção de falhas, principal método utilizado para a avaliação da dependabilidade de sistemas distribuídos.

2.2 Dependabilidade e Injeção de Falhas

O termo **dependabilidade** (*dependability*) vem substituindo o termo **tolerância a falhas**, e diz respeito a área de pesquisa ocupada com o comportamento de sistemas computacionais

sujeitos a ocorrência de falhas. A terminologia empregada na área de dependabilidade e que é utilizada neste trabalho segue o apresentado por *Avizienis et al.* em [Avizienis et al., 2004].

Definição 2.2.1 (Falha) *Uma falha (fault) é um comportamento incorreto externo ou interno ao sistema e que é a causa de um erro no sistema.*

Sistemas distribuídos são suscetíveis a diversos tipos de falhas, que geralmente são classificadas em falhas físicas, compreendendo as falhas de seus componentes, e as falhas humanas, compreendendo falhas de projeto e de implementação. Os modelos de falhas clássicos para sistemas distribuídos envolvem falhas que afetam a troca de mensagem, tais como: interrupção (*crash*), omissão (de envio ou recepção), temporização (resposta adiantada ou atrasada), integridade (respostas incorretas), e arbitrária (*byzantine*).

Definição 2.2.2 (Dependabilidade) *A dependabilidade de um sistema é a capacidade que o sistema tem para evitar interferências e interrupções no seu funcionamento que sejam mais frequentes do que o aceitável [Avizienis et al., 2004].*

A dependabilidade é um conceito integrador formado pelo conjunto de atributos listados a seguir.

— A disponibilidade (*availability*), que é a habilidade do sistema estar pronto para o acesso ao serviço correto.

— A confiabilidade (*reliability*), que é a continuidade do acesso à este serviço.

— A segurança (*safety*), que é a garantia de não haver consequências catastróficas ao usuário ou ao ambiente.

— A integridade (*integrity*), que é a ausência de alterações impróprias do funcionamento do sistema.

— A manutenibilidade (*maintainability*), que é a habilidade do sistema ser reparado e modificado sem interrupções.

O mecanismo de tolerância a falhas é o responsável por manter o sistema em funcionamento mesmo na presença de falhas. A injeção de falhas é uma técnica utilizada para avaliar a dependabilidade de um sistema considerando o seu mecanismo de tolerância a falhas. Essa técnica consiste em executar um sistema em um ambiente controlado, injetar falhas, e observar o comportamento do sistema [Hsueh et al., 1997, Voas, 1997].

As abordagens para a injeção de falhas são classificadas por serem aplicadas: (1) no modelo do sistema em teste, (2) no hardware utilizado pelo sistema, (3) no sistema em si, ou ainda (4) de forma híbrida simulando tanto as falhas de hardware quanto as de software através de uma camada de software adicional [Echtle and Leu, 1994, Hsueh et al., 1997, Arlat et al., 2003]. O interesse deste trabalho é na abordagem para a injeção de falhas híbridas, também chamada injeção de falha implementada por software (*Software-Implemented Fault Injection* - SWIFI), pois possibilita lidar melhor com problemas de controle e repetição dos experimentos.

Um experimento de injeção de falhas compreende as informações necessárias para a completa execução do sistema enquanto as falhas são injetadas. Diversas ferramentas de injeção de falhas estão disponíveis para auxiliar e automatizar a realização de experimentos, como o FIONA [Jacques-Silva et al., 2006], CloudVal [Pham et al., 2011], ou ainda o Loki [Lefever et al., 2004]. Antes de escolher uma ferramenta é necessário especificar a carga de falhas e a carga de trabalho que se deseja utilizar durante o experimento. A carga de falhas consiste na especificação de quais falhas devem ser injetadas e as suas respectivas configurações, tais como: quando injetar, onde injetar e a duração das falhas. A carga de trabalho consiste na especificação das configurações e comandos necessários para a execução do sistema de acordo com a necessidade do experimento.

Quando um experimento de injeção de falhas é utilizado para validar um sistema considerando o seu mecanismo de tolerância a falhas, ele também é chamado de um caso de falha [Echtle and Leu, 1994, Ambrosio et al., 2007, Marynowski, 2013, Marynowski et al., 2013]. Um caso de falha pode ser visto como uma extensão de um caso de teste envolvendo um experimento de injeção de falhas.

A próxima seção descreve em mais detalhes o sistema MapReduce apresentando o seu mecanismo de tolerância à falhas. O objetivo é identificar os modelos de falhas suportados e assim, especificar tanto a carga de falhas quanto a carga de trabalho necessárias para a definição dos casos de falha, que por sua vez são requisitos para a realização do teste de dependabilidade desses sistemas.

2.3 MapReduce e a Tolerância a Falhas

MapReduce (MR) é um sistema para o processamento de dados de grande escala e foi apresentado em 2004 por Jeffrey Dean e Sanjay Ghemawat [Dean and Ghemawat, 2004]. O sistema apresenta um modelo de programação simplificado que torna transparente ao usuário o tratamento de eventos relacionados à execução de uma tarefa em um ambiente distribuído, tais como a distribuição do processamento e a tolerância a falhas. A tarefa é decomposta em um conjunto de sub-tarefas e o conjunto massivo de dados em porções menores, tal que cada sub-tarefa processa uma porção diferente do dado de forma paralela.

Seu ambiente de programação é baseado em duas funções de alto nível programadas pelo usuário chamadas: *Map* e *Reduce*. Cada uma dessas funções possui um comportamento bem definido, mas trabalham juntas. A função *Map* faz o processamento inicial do dado de entrada, mapeando o dado que está distribuído, e o seu resultado é manipulado posteriormente pela função *Reduce*, agrupando o dado e gerando o resultado final.

Um sistema MapReduce é composto por três subsistemas:

1. um sistema de arquivos distribuído (*Distributed File System* - DFS),
2. uma implementação MapReduce, e
3. uma aplicação MapReduce.

Em relação ao primeiro item, um sistema de arquivo distribuído divide, distribui e replica o dado de entrada nas máquinas de um *cluster*[1]. O GFS [Ghemawat et al., 2003] e

[1]Um grande conjunto de máquinas interligadas em rede.

o HDFS [Apache, 2012] são exemplos de um sistema de arquivo distribuído. Já uma implementação MapReduce possibilita a paralelização e a distribuição automática de uma aplicação MapReduce através das máquinas. O Hadoop [Apache, 2013c] e o Skynet [Pisoni, 2012] são exemplos de implementações MapReduce. Por fim, uma aplicação MapReduce é desenvolvida pelo usuário e implementa as funções *Map* e *Reduce* capazes de realizar um processamento desejado.

Os pseudocódigos mostrados no Algoritmo 2 e 3, formam o exemplo de uma aplicação MapReduce denominada *WordCount*. O objetivo dessa aplicação é contar as palavras distintas existente em um conjunto de caracteres.

O Algoritmo 2 descreve a função *Map*, que recebe como entrada um conjunto de caracteres chamado $arquivo$, identifica cada palavra w existente no $arquivo$, e associa o valor 1 para cada palavra encontrada. O Algoritmo 3 descreve a função *Reduce*, que recebe como entrada uma palavra e uma lista de contadores desta palavra, soma os valores existentes nessa lista, e escreve a palavra e o resultado da soma.

Algoritmo 2: Map

Entrada: $arquivo$, conjunto de caracteres.
Início
  Para cada $w \in arquivo$ **faça**
    $Associa(w, 1)$;
  fim
fim

Algoritmo 3: Reduce

Entrada: $palavra$, conjunto de caracteres; $valores$, lista de contadores.
Dados: $soma$, número inteiro.
Início
  $soma \leftarrow 0$
  Para cada $v \in valores$ **faça**
    $soma \leftarrow soma + v$
  fim
  $Escreve(palavra, soma)$
fim

A execução de uma aplicação MapReduce é realizada a partir da inicialização do DFS e da

implementação MapReduce nas máquinas desejadas, e da inserção do dado de entrada no DFS. A Figura 2.2 apresenta o fluxo de dados e de controle da execução de uma aplicação por um sistema MapReduce. O processo inicia com o componente *Master* recebendo uma aplicação MapReduce. O *Master* coordena a execução, decidindo quantas tarefas de *Map* e *Reduce* serão executadas e alocando-as para os componentes *Worker* disponíveis. Um componente *Worker* é um processo cliente da implementação MapReduce, e normalmente cada *Worker* é executado em uma máquina do *cluster*. O *Master* aloca *Workers* ociosos e associa uma tarefa *Map* ou *Reduce* para cada um. Por exemplo, na Figura 2.2, tanto o *Worker1* quanto o *Worker2* executam um *Map*, e depois um *Reduce*. Um *Worker* que é associado a uma tarefa *Map* lê a porção da entrada correspondente a ele a partir do DFS, aplica a função *Map* e escreve o resultado em arquivos locais. As localizações destes arquivos são informadas ao *Master* que é responsável por encaminhá-las aos *Workers* que executarão o *Reduce*. O *Worker* que executará a função *Reduce* lê estes arquivos remotamente a partir dos *Workers* que executaram o *Map*, aplica a função *Reduce* e escreve o resultado em um arquivo no DFS.

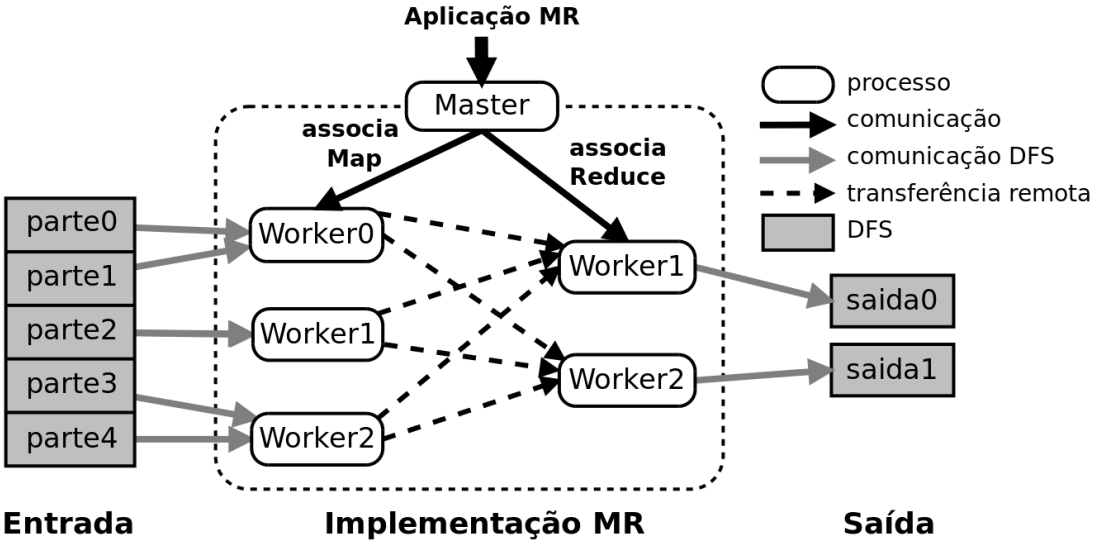


Figura 2.2: Visão geral da execução de um sistema MapReduce, baseada em [Dean and Ghemawat, 2004, Yang et al., 2007].

Como um sistema MapReduce é desenvolvido para utilizar milhares de máquinas, ele também é projetado para tolerar naturalmente falhas em algumas máquinas. Falhas de interrupção (*crash*) são toleradas pelo mecanismo de tolerância a falhas do MapReduce. A detecção é

feita pelo tempo de resposta (*timeout*), onde uma falha de um *Worker* é identificada e as suas tarefas são encaminhadas para um outro *Worker* ativo. Entretanto, como toda a coordenação é centralizada pelo *Master*, se ocorrer uma falha nele, a aplicação MapReduce é interrompida e deve ser reiniciada pelo usuário.

O processo de detecção e tratamento das falhas ocorridas nos *Workers* é definido da seguinte forma. O *Master* verifica periodicamente todo *Worker*, e se não obtém uma resposta ele marca este *Worker* como falho. Se um *Worker* falha enquanto executa uma tarefa *Map*, essa tarefa é reiniciada pelo *Master* e encaminhada para reexecução em outro *Worker* que contém a réplica do dado existente no *Worker* falho. De forma similar, se um *Worker* falha enquanto executa uma tarefa *Reduce*, essa tarefa também é reiniciada pelo *Master* e encaminhada para reexecução por um outro *Worker*. Entretanto, se um *Worker* falha após executar uma tarefa *Reduce*, esta tarefa não precisa ser reexecutada, pois seu resultado está gravado no sistema de arquivo distribuído.

Um tratamento diferenciado pelo mecanismo de tolerância a falhas ocorre quando um *Worker* X está executando uma tarefa *Reduce* e um outro *Worker* Y falha. Se o *Worker* Y contiver o resultado de uma tarefa *Map*, essa tarefa é reiniciada e encaminhada para reexecução por um *Worker* Z ativo, e o *Worker* X, que está executando a tarefa *Reduce*, é notificado de que precisa refazer a leitura do resultado a partir do *Worker* Z, pois o *Worker* Y não está mais acessível.

Existem trabalhos relacionados com a extensão dos sistemas MapReduce para que tolerem diferentes tipos de falha [Bessani et al., 2010, Costa et al., 2011, Lin et al., 2010] e também falhas do *Master* [Wang et al., 2013, White, 2012]. Embora a abordagem apresentada nesta tese tem o objetivo de ser extensível para o teste de sistemas que apresentem outros mecanismos de tolerância a falhas, o estudo e o teste foram restringidos ao mecanismo originalmente especificado pelo MapReduce. Em outras palavras, o estudo e o teste foram restringidos ao mecanismo que tolera falhas de interrupção de um *Worker* nas diferentes etapas de processamento, conforme descrito abaixo. O objetivo dessa restrição é facilitar o entendimento da nossa abordagem ao ser aplicada no sistema com o mecanismo de tolerância bem conhecido

e utilizado, e que ainda necessita ser testado.

2.4 Teste de Dependabilidade de um Sistema MapReduce

Realizar o teste de dependabilidade de um sistema MapReduce não é uma tarefa trivial. É necessário validar o comportamento do sistema mediante falhas, e o objetivo do teste de dependabilidade é encontrar erros no sistema, considerando principalmente o seu mecanismo de tolerância a falhas. Dois problemas principais são identificados para realizar o teste de dependabilidade de um sistema MapReduce. O primeiro problema é a geração de um conjunto de casos de falha representativos, e o segundo é desenvolver uma plataforma de teste que automatize a execução desses casos de falha. Ambos os problemas são descritos em mais detalhes a seguir.

2.4.1 Geração de Casos de Falha

Os trabalhos relacionados com a dependabilidade de sistemas MapReduce, atribuem a escolha e geração dos casos de falha ao engenheiro de teste. O engenheiro de teste escolhe e gera os casos de falha que ele considera representativos para o teste, baseando-se no seu conhecimento do sistema e do mecanismo de tolerância a falhas. Entretanto, os trabalhos assim relacionados descrevem apenas alguns casos de falha e que não consideram as etapas de processamento, sendo insuficientes para garantir a dependabilidade dos sistemas MapReduce.

Dean e Ghemawat [Dean and Ghemawat, 2004], autores do MapReduce, garantem a tolerância a falhas deste sistema aplicando apenas dois casos de falha. No primeiro, os autores removem grupos de 80 máquinas, de quase 1800 máquinas, durante alguns minutos. No segundo, eles interrompem 200 dos também quase 1800 processos, simulando falhas nas máquinas. Em ambos os casos de falha, os resultados obtidos são iguais ao esperado, e assim os autores afirmam que a implementação original do MapReduce é tolerante a falhas.

O Hadoop [Apache, 2013c] é uma implementação de código aberto do MapReduce. Abouzeid et al. [Abouzeid et al., 2009] apresentam um sistema gerenciador de banco de dados

paralelo baseado no Hadoop chamado HadoopDB. A efetividade da tolerância a falhas do HadoopDB é validada a partir da execução de um único caso de falha e comparado o seu comportamento com o Hadoop. Este caso de falha consiste em interromper uma máquina, de um total de dez, quando metade da tarefa de *Map* tiver sido executada. Como o resultado obtido foi o esperado tanto com o HadoopDB quanto com o Hadoop, os autores garantem que o HadoopDB tem a mesma efetividade na tolerância a falhas quanto o Hadoop.

Sangroya et al. [Sangroya et al., 2012a, Sangroya et al., 2012b] apresentam o MRBS (*MapReduce Benchmark Suite*), que é um conjunto de testes padronizados, ou *benchmarks*, para a avaliação da dependabilidade de sistemas MapReduce. O MRBS envolve a definição e a geração tanto de cargas de trabalho quanto de falhas. Um usuário gera os casos de falhas informando a carga de trabalho, quais os tipos de falha, e o momento e o período em que essas falhas serão injetadas. Para avaliar a tolerância a falhas do Hadoop, eles executam um caso de falha que consiste na interrupção de 100 processos *Map*, 5 minutos após o inicio do processamento, mas não especificam a quantidade total de processos existente. Após 25 minutos, 3 máquinas são interrompidas, de um total de 10 máquinas. Embora o MRBS seja flexível e possibilite a criação de novos casos de falha, ele ainda é restritivo, pois não possui um controle individualizado dos componentes e disponibiliza apenas o controle temporal para a injeção das falhas.

Os trabalhos relatados permitem avaliar alguns atributos da dependabilidade dos sistemas MapReduce, porém ainda são insuficientes para garantir a tolerância a falhas desses sistemas. Os casos de falha gerados e utilizados não consideram a injeção de falhas nas diferentes etapas de processamento de cada componente, característica fundamental do mecanismo de tolerância a falhas.

A maioria dos trabalhos que empregam a técnica de injeção de falhas para a avaliação ou validação de outros sistemas distribuídos, também atribuem a escolha dos casos de falha para o engenheiro de teste [Bernardi et al., 2012, Jacques-Silva et al., 2006, Lefever et al., 2004], ou geram os casos de falha de forma aleatória [Benso et al., 2007, Chandra et al., 2007, Team, 2012, Henry, 2009]. Deixar a geração dos casos de falha a cargo do engenheiro

de teste faz com que o teste seja limitado ao seu conhecimento. Os trabalhos que abordam a geração de casos de falha de forma aleatória buscam suprir essa deficiência. Porém, esses trabalhos também podem não encontrar erros que se manifestam apenas quando é injetada uma sequência específica de falhas.

Gunawi et al. [Gunawi et al., 2010, Gunawi et al., 2011, Gunawi, 2009] e Avresky et al. [Avresky et al., 1996] apresentam a geração dos casos de falha sistematicamente a partir do código fonte. Os casos de falha são gerados de forma exaustiva, inserindo uma falha em todos os pontos do código possíveis. Mesmo aplicando técnicas de poda [Joshi et al., 2011, Fu et al., 2004, Marinescu et al., 2010, Joshi, 2012], essas abordagens ainda são muito custosas. Além do mais, os casos de falha gerados não levam em consideração a etapa de processamento dos componentes e são limitados a um baixo número de falhas concorrentes ou sequenciais.

Os casos de falha também podem ser gerados a partir da especificação ou de uma abstração do mecanismo de tolerância a falhas implementado pelo sistema em teste. Ambrosio et al. [Ambrosio et al., 2005, Ambrosio, 2005] apresentam uma metodologia para a geração de casos de falha para o teste de sistemas espaciais. Os casos de falha são gerados a partir do modelo do mecanismo de tolerância a falhas descrito usando uma máquina de estados finita [Ural, 1992] e diagramas de sequência.

Bernardi et al. [Bernardi et al., 2012] apresentam uma abordagem para a modelagem e a avaliação da dependabilidade de sistemas desenvolvidos utilizando UML (*Unified Modeling Language* - Linguagem de Modelagem Unificada) [Group, 2013]. Eles propõem a UML-SM (UML *State Machine* - máquina de estados em UML), que é uma extensão da UML e possibilita a especificação do mecanismo de tolerância a falhas de um sistema de votação. A avaliação é feita do modelo do sistema e não do próprio sistema. A UML-SM é transformada em uma rede de Petri [Murata, 1989] que é analisada para verificar se a modelagem da tolerância a falhas está correta.

Echtle e Leu [Echtle and Leu, 1994] apresentam uma abordagem para a validação dos mecanismos de tolerância a falhas existentes nos protocolos de comunicação. A geração de casos de falha é feita a partir de uma análise da alcançabilidade do modelo do mecanismo em

redes de Petri. Cada caminho no grafo de alcançabilidade obtido da rede de Petri é um caso de falha que precisa ser executado.

Os casos de falha gerados a partir de um modelo formal do mecanismo de tolerância a falhas de um sistema, podem ser considerados representativos para o teste de dependabilidade. O teste é realizado considerando os casos de falha que devem ser tolerados pelo sistema, ao invés de testar todos os casos de falha possíveis em um ambiente distribuído. Entretanto, para aplicar essa abordagem para o teste de dependabilidade do MapReduce, é necessário criar um modelo formal adequado ao seu mecanismo de tolerância a falhas, além de indicar o processo para a geração dos casos de falha a partir desse modelo.

O modelo formal deve ser capaz de representar a dinâmica de um sistema MapReduce. Os seus componentes distribuídos devem ser modelados para que sejam facilmente inseridos e removidos. Essa característica possibilita representar as falhas toleradas, mas também é necessário representar o comportamento individual e dinâmico de cada componente. Os componentes executam tarefas em paralelo e uma tarefa pode ser executada por qualquer componente habilitado.

2.4.2 Automatização da Execução de Casos de Falha Representativos

Uma plataforma de teste é um sistema capaz de realizar a automatização da execução de casos de falha representativos para o teste de dependabilidade de sistemas MapReduce. A plataforma deve controlar e monitorar individualmente cada um dos componentes MapReduce distribuídos em diversas máquinas, injetar falhas de acordo com as suas etapas de processamento, e validar se o comportamento do sistema está de acordo com o especificado.

Sangroya et al. propuseram em [Sangroya et al., 2012a, Sangroya et al., 2012b] uma plataforma que executa cargas de trabalho e de falhas para avaliação da dependabilidade de sistemas MapReduce. A plataforma automatiza a execução de carga de falhas envolvendo diferentes tipos de falhas e em diferentes taxas. Entretanto, a plataforma não injeta falhas de

acordo com as etapas de processamento dos componentes e não valida o comportamento do MapReduce.

MRUnit [Apache, 2013a] e Herriot [Apache, 2013b] são um conjunto de bibliotecas que provêm interfaces para validar pequenas partes do sistema, por exemplo, um método ou uma função. Herriot foi proposto para ser uma plataforma de teste em grande escala, mas o seu desenvolvimento foi interrompido em uma versão que controla apenas testes de pequenas partes como o MRUnit.

Csallner et al. em [Csallner et al., 2011] apresentam uma plataforma de teste para aplicações MapReduce. Os autores apresentam um conjunto de regras de corretude e propõem uma busca sistemática por funções *Map* e *Reduce* que não seguem essas regras. Essas funções mal definidas podem ser identificadas quando ocorrem falhas em alguns componentes, e que altere a ordem em que o dado de entrada é processado. Por exemplo, uma função desenvolvida mal definida para retornar os 100 maiores elementos, mas que não considera todos os elementos, retornando os 100 primeiros maiores que outros. Essa plataforma não injeta falhas, não controla a execução dos casos de falha e nem valida o comportamento do sistema.

A plataforma Ganesha [Pan et al., 2010] é uma solução usada para detectar problemas de performance em sistemas MapReduce. A avaliação é feita a partir dos históricos de execução (*logs*), levando em consideração as informações do sistema operacional e as da rede. A plataforma não requer modificações no MapReduce, produz baixa sobrecarga, e possibilita identificar máquinas que estão comprometendo o desempenho do sistema. Entretanto, sua avaliação ocorre apenas após a execução e não controla os componentes do MapReduce durante a execução dos experimentos.

Outros trabalhos também avaliam a execução de sistemas MapReduce por análise de log para identificar problemas de performance [Tan et al., 2008, Pan et al., 2009, Tan et al., 2009, Huang et al., 2010]. Embora essas abordagens avaliem a funcionalidade e a performance de sistemas MapReduce, elas não são capazes de testar a dependabilidade destes sistemas. Essas abordagens não injetam falhas, não controlam os componentes do MapReduce, e nem validam o seu comportamento.

Algumas plataformas de teste para sistemas P2P [Androutsellis-Theotokis and Spinellis, 2004] apresentam soluções para o controle dos componentes distribuídos e para a validação do comportamento do sistema em teste [Butnaru et al., 2007, Zhou et al., 2006, de Almeida et al., 2010b]. Entretanto, essas plataformas não consideram a injeção de falhas e nem consideram as etapas de processamento dos componentes e do sistema em teste. Ainda assim, a plataforma de teste PeerUnit [de Almeida et al., 2010c, de Almeida et al., 2010b] se destaca nesse conjunto automatizando a execução de casos de teste distribuídos, que são descritos facilmente e coordenados de forma otimizada [de Almeida et al., 2010a].

Outras plataformas empregadas para a injeção de falhas em sistemas distribuídos suportam a execução de casos de falha com a injeção de diversas e múltiplas falhas, mas não controlam a execução do sistema [Jacques-Silva et al., 2006, Pham et al., 2011, Stott et al., 2000, Lefever et al., 2004, Hoarau et al., 2007]. Essa característica impossibilita identificar dinamicamente em quais componentes e em quais etapas de processamento as falhas devem ser injetadas. Além disso, nenhuma das plataformas citadas apresenta resultados sobre a dependabilidade dos sistemas MapReduce.

2.5 Considerações

Existem diversos tipos de teste de software, entre eles, o teste de instalação, compatibilidade, funcionalidade, usabilidade, e internacionalização. Este trabalho objetiva definir e realizar o teste de dependabilidade de um sistema MapReduce, ou seja, validar o seu comportamento na presença de falhas.

Dois grandes problemas surgem para realizar o teste de dependabilidade. O primeiro é gerar um conjunto de casos de falha que sejam importantes para revelar erros no sistema considerando o seu mecanismo de tolerância. O segundo é automatizar a execução desses casos de falha em um ambiente real de execução do sistema.

Quatro abordagens foram apresentadas para a geração de casos de falha: a geração pelo engenheiro de teste, de forma aleatória, por busca sistêmica, e a partir do modelo formal.

Embora a geração pelo engenheiro de teste e a de forma aleatória sejam empregadas para a avaliação de sistemas MapReduce, essas abordagens são insuficientes para garantir a tolerância a falhas desses sistemas. Os casos de falha gerados não consideram a injeção de falhas nas diferentes etapas de processamento de cada componente, característica fundamental do mecanismo de tolerância a falhas. A abordagem para gerar os casos de falha realizando uma busca sistêmica no código fonte do sistema, é muito custosa e limitada, mesmo aplicando técnicas de poda.

Os casos de falha gerados a partir de um modelo formal do mecanismo de tolerância a falhas podem ser considerados os mais representativos para o teste de dependabilidade. O teste é realizado considerando os casos de falha que devem ser tolerados pelo sistema, ao invés de testar todos os casos de falha possíveis em um ambiente distribuído. Entretanto, para aplicar essa abordagem para o teste de dependabilidade do MapReduce, é necessário criar um modelo formal adequado ao seu mecanismo de tolerância a falhas, além de indicar o processo para a geração dos casos de falha a partir desse modelo.

O modelo formal deve ser capaz de representar a dinâmica de um sistema MapReduce. Os seus componentes distribuídos devem ser modelados para que sejam facilmente inseridos e removidos. Essa característica possibilita representar as falhas toleradas, mas também é necessário representar o comportamento individual e dinâmico de cada componente. Os componentes executam tarefas em paralelo e uma tarefa pode ser executada por qualquer componente habilitado.

O próximo problema para o teste de dependabilidade de um sistema MapReduce é a automatização da execução dos casos de falha representativos. A plataforma de teste deve controlar e monitorar individualmente cada um dos componentes MapReduce distribuídos em diversas máquinas, injetar falhas de acordo com as suas etapas de processamento, e validar se o comportamento do sistema está de acordo com o especificado. Diversas plataformas foram estudadas, entretanto não suprem essa necessidade.

As plataformas de teste relacionadas com sistemas MapReduce não validam o comportamento do sistema em teste e não injetam as falhas de acordo com as etapas de processamento

dos componentes MapReduce. As plataformas empregadas para a injeção de falhas considerando outros sistemas distribuídos suportam a execução de casos de falha com a injeção de diversas e múltiplas falhas, mas não controlam a execução do sistema. Essa característica impossibilita identificar dinamicamente em quais componentes e em quais etapas de processamento as falhas devem ser injetadas. Além disso, nenhuma das plataformas citadas apresenta resultados sobre a dependabilidade dos sistemas MapReduce.

Outras plataformas realizam o teste de funcionalidade controlando os componentes distribuídos do sistema em teste e validam o seu comportamento. Por outro lado, essas plataformas não injetam falhas e não controlam o processamento de cada componente para que as falhas sejam injetadas.

Este trabalho apresenta uma abordagem para o teste de dependabilidade de um sistema MapReduce. Essa abordagem envolve a geração de casos de falha representativos a partir de um modelo formal, e a apresentação de uma plataforma de teste para a automatização da execução de casos de falha.

O próximo capítulo deste trabalho apresenta a modelagem do mecanismo de tolerância a falhas do MapReduce, e um processo para gerar casos de falha representativos a partir do modelo adotado. Na sequência, o capítulo 4 seguinte apresenta a plataforma de teste capaz de automatizar a execução de casos de falha em um ambiente distribuído.

CAPÍTULO 3

GERAÇÃO DE CASOS DE FALHA REPRESENTATIVOS

A geração de casos de falha representativos é o primeiro problema encontrado quando se deseja testar a tolerância a falhas de um sistema distribuído. Esse problema envolve escolher elementos representativos de um conjunto infinito e parcialmente desconhecido de cenários de falha que podem ocorrer em um ambiente distribuído [Voas, 1997].

Este capítulo está divido em três seções e apresenta, respectivamente, os principais fundamentos para a geração de casos de falha representativos, a modelagem do mecanismo de tolerância que resulta em um modelo formal representado em redes de Petri, e o processo para a geração dos casos de falha representativos gerados a partir do modelo formal.

3.1 Definições

Um caso de falha estende um caso de teste distribuído adicionando a injeção de falhas durante a execução do sistema em teste (*System Under Test* - SUT). Um caso de falha se diferencia de um experimento de falha, como proposto pela literatura de injeção de falhas. Um experimento de falha tem como objetivo avaliar o comportamento do sistema mediante falhas. Entretanto, um caso de falha tem como objetivo validar o comportamento do sistema mediante falhas, ou ainda encontrar erros no sistema em teste considerando o seu mecanismo de tolerância a falhas.

Assim como um caso de teste distribuído, um caso de falha envolve os componentes necessários para uma completa execução e validação de um SUT, e gera como resultado a validação se o sistema passou no teste, se falhou apresentando um erro, ou se foi inconclusivo impossibilitando afirmar qualquer um dos resultados anteriores. A diferença entre um caso de teste distribuído e um caso de falha é que as ações que compõem um caso de falha envolvem a injeção de falhas, e essa deve ocorrer de acordo com a etapa de processamento de cada

componente do sistema.

Definição 3.1.1 (Caso de Falha) *Um caso de falha é a quádrupla* $\mathcal{F} = (T^{\mathcal{F}}, A^{\mathcal{F}}, R^{\mathcal{F}}, \mathcal{O})$ *onde:*

- $T^{\mathcal{F}} = \{t_0, t_1, \ldots, t_n\}$ é uma lista de testadores que controlam os componentes do SUT;
- $A^{\mathcal{F}} = \{a_0, a_1, \ldots, a_m\}$ é uma lista de ações que envolve a injeção de falhas;
- $R^{\mathcal{F}} = \{r_{a_0}, \ldots, r_{a_m}\}$ é uma lista de resultados das ações;
- $\mathcal{O}$ é um oráculo.

O oráculo é o mecanismo responsável por verificar se o sistema comportou-se corretamente durante a execução de um caso de falha. O oráculo atribui o resultado da execução de um caso de falha, isto é, o veredito, que pode ser: *PASSOU*, *FALHOU* ou *INCONCLUSIVO*.

O veredito de um caso de falha é formado pelo resultado obtido de cada testador na execução de cada ação. Cada execução de uma ação a_i pode ter como resultado

$$r_{a_i} = (SUCESSO \oplus FRACASSO \oplus TIMEOUT),$$

onde $\oplus$ representa a operação lógica binária "*ou exclusivo*".

O resultado da execução de uma ação r_{a_i} é *SUCESSO* quando a execução de uma ação ocorre sem a presença de falha. O resultado é *FRACASSO* quando a execução de uma ação não pode ser realizada. O resultado é *TIMEOUT* quando não houve resposta da execução de uma ação após um limite de tempo. Se todos os resultados das execuções das ações, $R^{\mathcal{F}}$, forem *SUCESSO*, então o veredito do caso de falha $\mathcal{F}$ é *PASSOU*. Caso exista alguma execução de uma ação que tenha obtido o resultado *FRACASSO*, então o veredito do caso de teste é *FALHOU*. Mas, se pelo menos o resultado da execução de uma ação for *TIMEOUT*, o veredito do caso de falha $\mathcal{F}$ é *INCONCLUSIVO*, tornando o teste impreciso para se aferir qualquer uma das afirmações anteriores. Nesta situação o caso de falha precisa ser reexecutado.

Definição 3.1.2 (Ação de um Caso de Falha) *Uma ação de um caso de falha é a sétupla* $a_i = (h, D, n, T', W, \theta, I)$ *onde:*

- $h \in \mathbb{N} \mid h \leqslant |A^{\mathcal{F}}|$ é uma ordem hierárquica na qual a ação a_i deve ser executada. Uma observação importante é que as ações com o mesmo h são executadas em paralelo.
- $D \subseteq A^{\mathcal{F}} \mid \forall a_j \in D : r_{a_j} = SUCESSO$ é um conjunto de ações que devem ser executadas antes de a_i, e devem obter $SUCESSO$ como resultado de suas execuções. Caso contrário, o resultado da execução da ação a_i, r_{a_i}, é $FRACASSO$.
- $n \in \mathbb{N} \mid n \leqslant |T'|$ é o número de testadores que devem executar a ação a_i e obter $SUCESSO$ como resultado.
- $T' \subseteq T^{\mathcal{F}}$ é um conjunto de testadores que podem executar a_i.
- W é um gatilho, isto é, uma instrução ou um comando opcional que é executável pelos testadores e é requerido para executar a_i;
- θ é um limite de tempo para executar a_i; e,
- I é uma instrução ou comando executável pelos componentes do SUT.

O limite de tempo θ é importante para garantir o término da execução do caso de falha, uma vez que podem ocorrer problemas na execução das ações fazendo com que o teste não finalize. O gatilho W é necessário para que a injeção de falhas ocorra em um determinado estado do SUT, ou de um de seus componentes.

A representatividade de um caso de teste é definida como a importância que este possui para identificar erros em um SUT [Arlat et al., 2003, Natella et al., 2012]. A representatividade de um caso de falha segue essa mesma definição e considera a dependabilidade do SUT.

Definição 3.1.3 (Caso de Falha Representativo) *Um caso de falha representativo é o caso de falha que tem uma grande possibilidade de identificar erros no SUT considerando a sua dependabilidade.*

Os casos de falha representativos para o teste de dependabilidade de um sistema MapReduce são os casos de falha gerados a partir do modelo do seu mecanismo de tolerância a falhas. O modelo formal desse mecanismo representa o comportamento do sistema mediante falhas. Os casos de falha extraídos desse modelo devem ser tolerados pelo sistema, e possuem uma grande possibilidade de identificar erros considerando a dependabilidade do sistema. Portanto, esses casos de falha são considerados representativos para o teste de dependabilidade do MapReduce.

A seção seguinte apresenta a modelagem do mecanismo de tolerância a falhas do MapReduce. O modelo resultante é utilizado para gerar os casos de falha representativos para o teste de dependabilidade de um sistema MapReduce, e cuja abordagem é apresentada na seção posterior.

3.2 Modelagem do Mecanismo de Tolerância a Falhas do MapReduce

Para modelar o mecanismo de tolerância a falhas de um sistema MapReduce é necessário o uso de um modelo formal (ou abstração) que possibilite representar os comportamentos concorrentes e distribuídos dos diferentes componentes do MapReduce, isto é, modelar o comportamento do *Master* e dos vários *Workers*. O modelo deve representar os componentes do MapReduce como itens dinâmicos, de modo que possam ser facilmente removidos ou inseridos, sem mudanças substanciais no modelo. Além disso, o modelo deve representar os componentes sem especificar suas ações, permitindo que uma ação seja executada por qualquer componente habilitado. Essa característica é importante para modelar o processo de reencaminhamento de tarefas de *Map* e *Reduce* falhas.

As principais abordagens para modelar os sistemas distribuídos envolvendo suas características de dependabilidade são a modelagem baseada em máquina de estados finitas (*Finite State Machine* - FSM) [Ural, 1992] e a modelagem baseada em redes de Petri (*Petri Net* - PN) [Echtle and Leu, 1994]. O modelo em FSM abstrai detalhes do funcionamento do sistema

ao mesmo tempo que representa características testáveis, como as saídas produzidas em cada estado do sistema. O modelo em PN possui um poder maior de representação e é utilizado para representar sistemas com características concorrentes, distribuídas, e não-determinísticas, tal como o MapReduce.

A seguir é apresentada a modelagem do mecanismo de tolerância a falhas do MapReduce utilizando máquina de estados finitas, e posteriormente a modelagem utilizando redes de Petri.

3.2.1 Modelagem Usando Máquinas de Estados Finitos

Uma máquina de estados finitos (*Finite State Machine* - FSM), ou simplesmente uma máquina de estados, é um modelo matemático usado para representar o comportamento de um sistema considerando seus estados, entradas e saídas [Ural, 1992]. Uma máquina de estados é formalmente definida como a quíntupla $FSM = (S, I, O, \delta, \lambda)$, onde:

— S é um conjunto finito de estados, sendo um deles o estado inicial;

— I é um conjunto finito de entradas;

— O é um conjunto finito de saídas;

— $\delta : S \times I \rightarrow S$ é uma função de transição que mapeia pares de estados e entradas com um próximo estado;

— $\lambda : S \times I \rightarrow O$ é uma função de transição que mapeia pares de estados e entradas com a saída correspondente.

As entradas e as saídas de uma máquina de estados podem ser interpretadas como eventos de um sistema, e o diagrama de estados em UML usa essa abordagem para representar visualmente uma FSM [Group, 2013]. Nesse diagrama o estado inicial é representado por um círculo preto todo preenchido e o estado final por um círculo com um círculo menor preenchido no seu interior. Os estados do sistema são representados por retângulos arredondados rotulados com identificadores desses estados. As transições são representadas por setas rotuladas com os eventos do sistema.

A Figura 3.1 mostra um diagrama de estados que representa, de forma simplificada, a execução de uma aplicação MapReduce. O estado *MRiniciado* representa que o sistema MapReduce já está iniciado, ou seja, o componente *Master* e os componentes *Worker* estão ativos. O próximo evento possível é o *iniciaAplicação* e representa que o componente *Master* inicia uma aplicação MapReduce para ser executada, como um *WordCount*. Seguindo o fluxo de execução do MapReduce, esse evento faz com que o sistema entre no estado *executandoMap*, que representa que a tarefa *Map* está em execução. O próximo evento possível é *iniciaReduce*, que faz com que o sistema passe para o próximo estado *executandoReduce*. A partir desse estado, o próximo evento possível é *finalizaAplicação*. Esse evento representa que a aplicação foi finalizada e o resultado se encontra disponível para utilização.

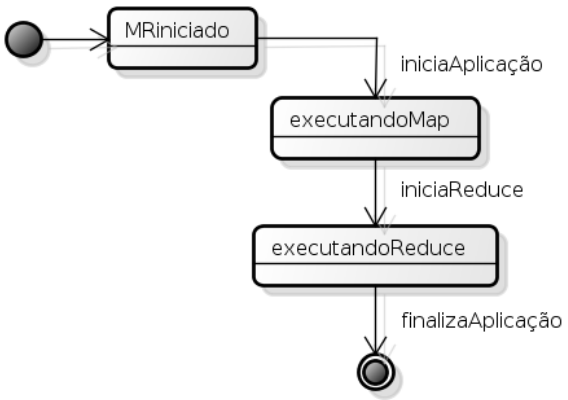


Figura 3.1: Diagrama de estados da execução de uma aplicação MapReduce.

Conforme apresentado na Seção 2.3 do capítulo anterior, o mecanismo de tolerância a falhas do MapReduce é baseado na reexecução, e possui um tratamento diferenciado dependendo da etapa de processamento em que uma falha ocorre. Uma FSM possibilita abstrair os detalhes do mecanismo de tolerância a falhas, permitindo fazer uma relação direta com as etapas de processamento do MapReduce, como o estado *executandoMap* e o evento *iniciaReduce* apresentados na Figura 3.1. Entretanto, os estados e eventos são diferentes para cada componente do MapReduce, por exemplo, o *WorkerX* executa uma tarefa de *Map* e o *WorkerY* executa uma tarefa de *Reduce*. Nesse sentido, os estados e as ações de cada componente foram representados usando a nomenclatura dos estados e eventos do diagrama

de estados.

A Figura 3.2 mostra um diagrama de estados que representa uma parte do mecanismo de tolerância a falhas do MapReduce. A parte modelada refere-se a manipulação das falhas ocorridas enquanto um *Worker* executa uma tarefa *Map* ou de *Reduce*. O estado *MRiniciado* indica que o sistema MapReduce está iniciado em três máquinas, uma com o componente *Master* e outras duas com os componentes *Worker* identificados como *WorkerX* e *WorkerY*. O próximo evento é o *Master.iniciaAplicação*. Esse evento é executado pelo componente *Master*, que recebe uma aplicação MapReduce, e prossegue a sua execução encaminhando as tarefas de *Map* para serem executadas. Então, o próximo estado é *WorkerX.executandoMap*, que representa que o *WorkerX* está executando uma tarefa *Map*. Esse estado possibilita a ocorrência de dois eventos. Ocorrendo o evento *WorkerX.iniciaReduce*, indica que a tarefa *Map* foi finalizada sem qualquer intervenção, e o sistema segue para o estado *WorkerX.executandoReduce*. O estado *WorkerX.executandoReduce* também possibilita a ocorrência de dois novos eventos. Ocorrendo o evento *Master.finalizaAplicação*, indica que a tarefa *Reduce* também foi finalizada sem intervenções, e o resultado da aplicação pode ser validado. Por outro lado, ocorrendo o evento *WorkerX.falha*, indica que o *WorkerX* falha enquanto executa uma tarefa *Reduce*.

A ocorrência do evento *WorkerX.falha* faz o sistema ir para o estado *WorkerY.executandoReduce*, que indica que a tarefa *Reduce* foi encaminhada para o *WorkerY* e está em execução. A partir do estado *WorkerY.executandoReduce* o evento possível também é *Master.finalizaAplicação*, que indica a finalização da aplicação e que o resultado pode ser validado. Voltando ao estado *WorkerX.executandoMap*, o outro evento possível é o *WorkerX.falha*. Ocorrendo esse evento, que indica que o *WorkerX* falha enquanto executa uma tarefa *Map*, o sistema vai para o estado *WorkerY.executandoMap*. Esse estado indica que a tarefa *Map* foi encaminhada para o *WorkerY* e está em execução. O evento possível no estado *WorkerY.executandoMap* é *WorkerY.iniciaReduce*, que indica que a tarefa *Map* foi finalizada e o sistema segue para o estado *WorkerY.executandoReduce*. Assim como descrito anteriormente, nesse estado o evento possível é o *Master.finalizaAplicação*, que indica que o resultado está disponível para ser validado.

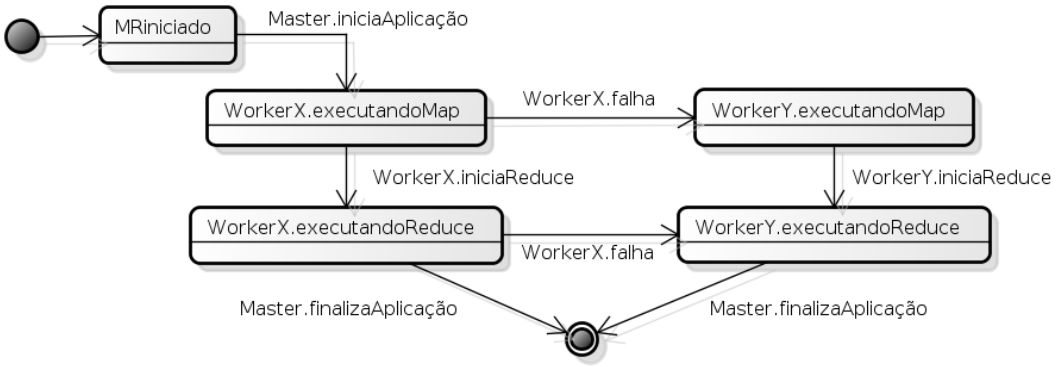


Figura 3.2: Diagrama de estados do mecanismo de tolerância a falhas do MapReduce com 3 componentes.

A modelagem apresentada consegue representar uma parte do mecanismo de tolerância a falhas. Entretanto, a modelagem limita a representação de comportamentos mais complexos, como ocorre ao considerar falhas recorrentes, ou com um número maior de componentes do MapReduce.

A Figura 3.3 mostra um diagrama de estados que representa a parte do mecanismo que trata a ocorrência de duas falhas. O estado *MRiniciado* representa agora que o sistema MapReduce está iniciado em quatro máquinas, uma com o componente *Master* e outras três com os componentes *Worker* identificados como *WorkerX*, *WorkerY* e *WorkerZ*. Adicionalmente ao apresentado no modelo anterior (Figura 3.2), o modelo atual representa que o *WorkerY* também pode falhar enquanto executa uma tarefa *Map* ou *Reduce*. Esses eventos são representados pela transição *WorkerY.falha* que pode originar tanto do estado *WorkerY.executandoMap* quanto do estado *WorkerY.executandoReduce*. Dois novos estados são adicionados: *WorkerZ.executandoMap* e *WorkerZ.executandoReduce*. Esses dois novos estados são ligados pelo evento *WorkerZ.iniciaReduce*, e que, na sequência, conduz para o resultado da aplicação.

Embora existam extensões para a representação de diversas características, como a representação do tempo [El-Fakih et al., 2009], o uso de FSM ainda é restritivo à modelagem de vários componentes que possuem comportamentos distintos e paralelos – que são requerimentos dos sistemas MapReduce. Cada componente MapReduce precisa de um conjunto de estados e eventos específicos para modelar o seu comportamento. Neste sentido, a próxima

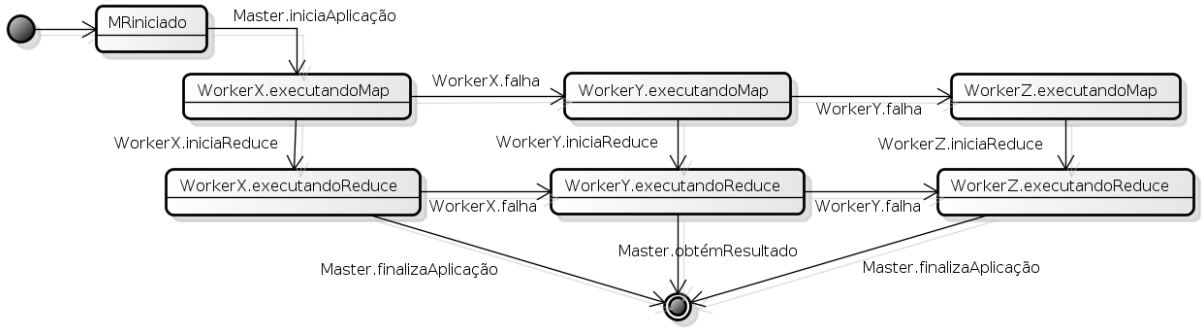


Figura 3.3: Diagrama de estados do mecanismo de tolerância a falhas do MapReduce com 4 componentes.

seção apresenta a modelagem usando redes de Petri, objetivando representar de forma mais adequada o mecanismo de tolerância a falhas do MapReduce.

3.2.2 Modelagem Usando Redes de Petri

Uma rede de Petri (*Petri Net* - PN) é um modelo formal utilizado para representar sistemas com características concorrentes, assíncronas, distribuídas, não-determinísticas e estocásticas [Murata, 1989, Peres, 2010, Callou et al., 2012]. Uma rede de Petri é formalmente definida pela quíntupla $PN = (P, T, F, W, M_0)$, onde:

— $P = \{p_1, p_2, ..., p_m\}$ é um conjunto finito de lugares;

— $T = \{t_1, t_2, ..., t_n\}$ é um conjunto finito de transições;

— $F \subseteq (P \times T) \bigcup (T \times P)$ é um conjunto finito de arcos formado pela relação entre os lugares e as transições, isto é, os arcos que ligam os lugares com as transições e vice-versa;

— $W : F \rightarrow \{1, 2, 3, ...\}$ é uma função de peso para os arcos; e,

— $M_0 : P \rightarrow \{0, 1, 2, 3, ...\}$ é uma marcação inicial.

Uma rede de Petri é graficamente representada por um grafo composto por círculos que representam os lugares, barras que representam as transições, e arcos que ligam os lugares com as transições e vice-versa. Os arcos podem possuir um número relacionado, e este representa o seu peso, que é 1 quando ausente. No interior dos lugares existem pequenos círculos pretos,

chamados de marcas, que são consumidos de um lugar e criados num outro lugar após o disparo de uma transição. Qualquer distribuição de marcas pelos lugares da rede representam um estado do sistema, e é chamado de uma marcação na rede de Petri. A transição só é disparada quando existem marcas suficientes nos lugares dos arcos de entrada conectados à referida transição.

Os componentes de uma rede de Petri são interpretados de acordo o sistema modelado. Usualmente, a abordagem empregada para a modelagem de sistemas distribuídos considera uma marca como uma mensagem ou um dado, que assume vários estados (lugares) e sofre algumas ações (transições).

A Figura 3.4 mostra o exemplo de uma rede de Petri que modela uma parte do sistema MapReduce usando a abordagem usual. A rede modela de forma simplificada o comportamento do *Master* e do *Worker* na execução de tarefas *Map*. Os lugares representam os estados dos componentes, as transições representam suas ações, e as marcas especificam os estados dos componentes e representam uma mensagem ou uma tarefa *Map*. Inicialmente, a transição *Master.associaMap* é disparada, pois é a única habilitada, e representa a ação do *Master* de associar uma tarefa *Map* para um *Worker*. A transição *Master.associaMap* consome uma marca do lugar *Master.temMapWorker*, e gera uma marca no lugar *rede*. Agora, a transição *Worker.executaMap* pode ser disparada, e representa a execução da tarefa *Map* pelo *Worker*. A transição *Worker.executaMap* consome uma marca do lugar *rede* e outra do lugar *Worker.pronto*, e gera uma marca novamente no lugar *Worker.pronto* e no lugar *Master.temMapWorker*. Assim, uma nova tarefa *Map* pode ser associada ao *Worker*, disparando a transição *Master.associaMap*, e assim por diante.

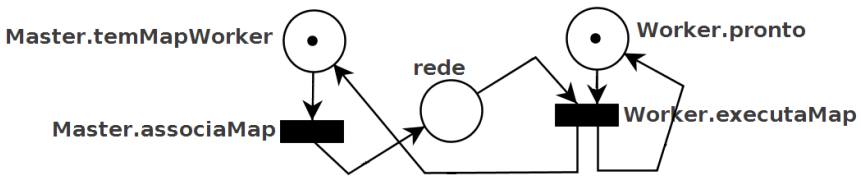


Figura 3.4: Rede de Petri de uma pequena parte do sistema MapReduce usando a abordagem usual.

A abordagem usual utilizada para modelar o mecanismo de tolerância a falhas do MapReduce confirmou o poder de representação das ações distintas e paralelas de um sistema distribuído. Entretanto, a abordagem não permitiu a modelagem da independência dos estados e ações com os componentes do sistema. Para representar um outro *Worker* o modelo também deve ser expandido de modo a gerar um conjunto específico de estados e ações para o novo *Worker*.

Este trabalho apresenta uma nova abordagem para interpretar os componentes de uma rede de Petri. Essa abordagem surgiu da abstração requerida para a modelagem do mecanismo de tolerância, e utilizada para a modelagem usando máquinas de estados finitos, apresentada na seção anterior.

Nesta nova abordagem, as marcas representam os componentes MapReduce, os lugares representam os estados ou as etapas de processamento desses componentes, e as transições representam as ações de um caso de falha. Dessa forma, os componentes MapReduce são modelados como itens dinâmicos, ou seja, os componentes são inseridos e removidos durante a execução de uma ação (ou disparo de uma transição).

Esta abordagem ainda permite modelar a independência dos componentes MapReduce com suas ações e estados, isto é, uma ação pode ser executada por qualquer componente habilitado. Um estado do sistema MapReduce considerando todos os seus componentes é representado por uma marcação na rede de Petri, isto é, uma distribuição específica de marcas pelos lugares.

A Figura 3.5 mostra uma rede de Petri que representa uma parte do mecanismo de tolerância a falhas do MapReduce. A parte modelada é referente à manipulação das falhas ocorridas enquanto um *Worker* executa uma tarefa *Map* ou de *Reduce*. A marcação inicial é composta por uma marca em *Master.ativo* e duas marcas em *Worker.ativo*, e representa que um componente *Master* e dois componentes *Worker* estão ativos. A marcação inicial permite apenas que a transição *Master.iniciaAplicação* seja disparada. Com o seu disparo, duas marcas são consumidas, uma do lugar *Master.ativo* e outra do lugar *Worker.ativo*, e um nova marca é produzida no estado *Worker.executandoMap*. Agora, duas transições podem disparar, *nada* e

Worker.falha.executandoMap. Se a transição *nada* dispara, ela consome uma marca de *Worker.executandoMap* e produz uma marca em *Worker.executandoReduce*. Se a transição *Worker.falha.executandoMap* dispara, ela consome uma marca do lugar *Worker.executandoMap* e uma do lugar *Worker.ativo*, e produz uma marca novamente em *Worker.executandoMap*. Este comportamento é similar se a transição *Worker.falha.executandoReduce* dispara quando existe uma marca em *Worker.executandoReduce*, embora também seja necessário que se tenha uma marca em *Worker.ativo*. Por fim, o disparo da transição *Master.finalizaAplicação* consome uma marca de *Worker.executandoReduce* e gera uma em *Worker.fim*, e representa que a aplicação foi finalizada e que o resultado está disponível.

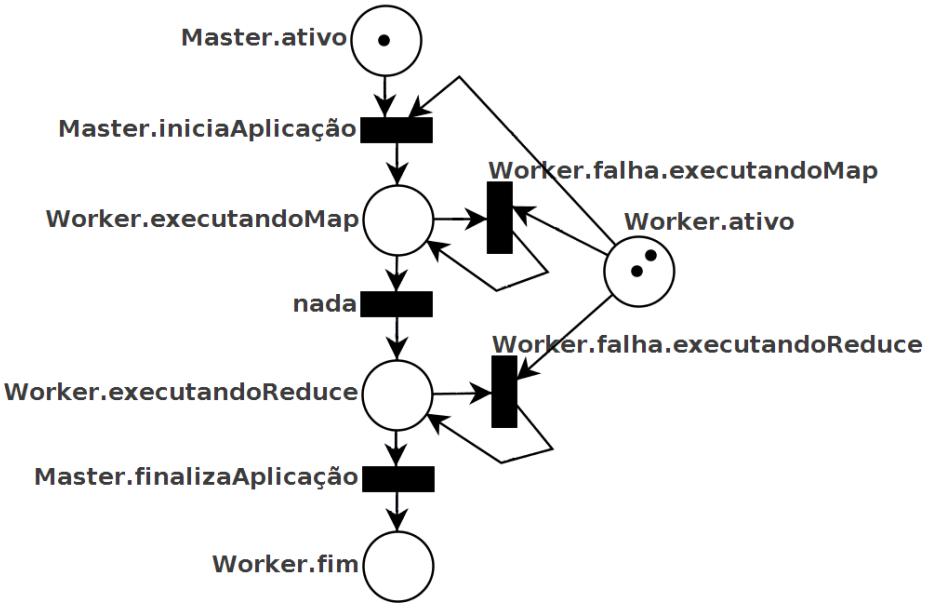


Figura 3.5: Rede de Petri do mecanismo de tolerância a falhas do MapReduce usando a nova abordagem.

Para aumentar ou diminuir o número de componentes *Worker* é necessário apenas adicionar ou remover marcas no lugar correspondente à *Workers.ativo*. Dessa forma, os componentes MapReduce são dinamicamente adicionados ou removidos do sistema, apenas alterando o número de marcas na rede. A modelagem de uma ação também é independente do componente que a executa. Qualquer componente disponível no estado requerido para ativar uma determinada ação, pode executar essa ação. Considerando a rede de Petri, tem-se que qualquer marca existente no lugar requerido para ativar determinada transição pode ativar essa transi-

ção. Assim, comprova-se o maior poder de representação da rede de Petri. A rede de Petri possibilita a modelagem dos estados e transições, como a máquina de estados, e possibilita ainda a modelagem dos vários componentes MapReduce dinamicamente e sem especificar as ações que cada um possui.

Além de permitir uma modelagem compreensiva do mecanismo de tolerância a falhas do MapReduce, a rede de Petri permite extensões do modelo. As extensões para rede de Petri permitem ainda modelar outros comportamentos implicitamente especificados no mecanismo de tolerância a falhas do MapReduce, tais como a identificação de falhas temporais e a interrupção quando é impossível completar uma tarefa.

A próxima seção apresenta uma abordagem para a geração dos casos de falha representativos, a partir do modelo em rede de Petri apresentado nesta seção.

3.3 Abordagem para a Geração de Casos de Falha Representativos

A representatividade de um caso de falha é a importância que este tem para identificar erros em um sistema. Para encontrar erros relacionados à dependabilidade de um sistema MapReduce é necessário testar se o sistema realmente suporta as falhas de acordo com o seu mecanismo de tolerância a falhas. O modelo deste mecanismo representa os cenários e as falhas toleradas pelo sistema. Assim, os casos de falha representativos para o teste de dependabilidade para um sistema MapReduce são os casos gerados a partir do modelo do mecanismo de tolerância a falhas deste sistema. Essa abordagem guia o processo de geração para um conjunto finito de casos de falha que devem ser tolerados pelo sistema, e que precisam ser validados para realizar o teste de dependabilidade do sistema.

Nesta seção apresentamos o processo empregado para a geração de casos de falha representativos para o teste de dependabilidade de um sistema MapReduce. O processo consiste na extração dos casos de falha a partir de um grafo de alcançabilidade da rede de Petri que modela o mecanismo de tolerância a falhas do sistema MapReduce. *Echtle & Leu* em [Echtle and Leu, 1994] utilizam um processo semelhante para o teste de protocolos de rede, onde a

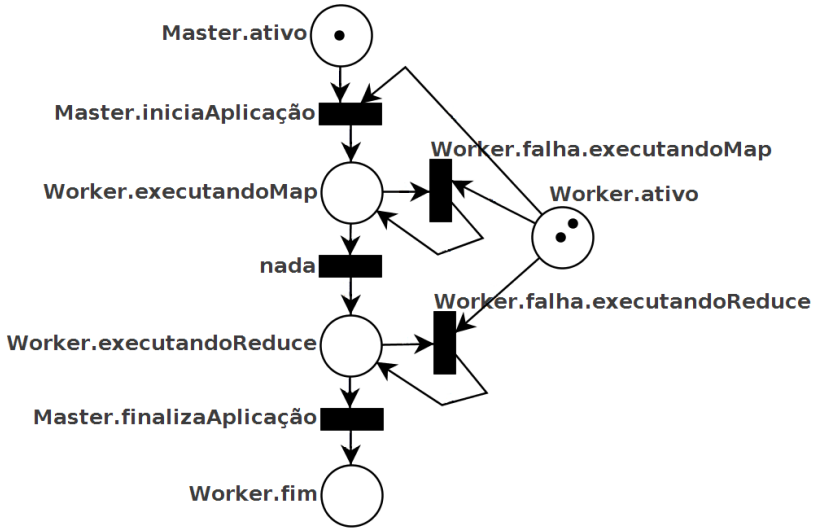
rede de Petri representa o comportamento de cada nó da rede mediante o tratamento das mensagens recebidas e encaminhadas.

3.3.1 O Grafo de Alcançabilidade de uma Rede de Petri

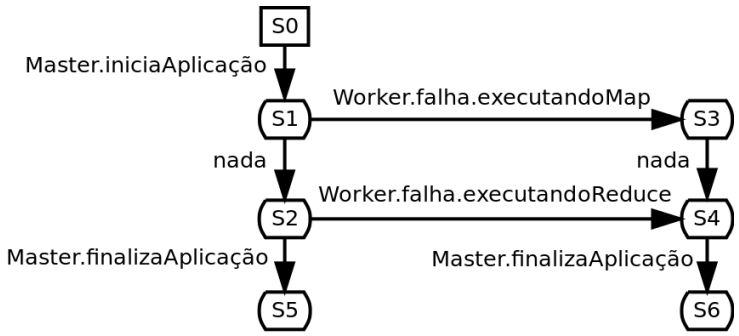
Um grafo de alcançabilidade consiste em um conjunto com todas as sequências possíveis de disparos das transições da uma rede de Petri [Echtle and Leu, 1994]. Os **vértices** do grafo representam as marcações da rede de Petri, e as **arestas** representam as transições executadas para que haja a mudança entre as marcações.

A Figura 3.6(a) mostra novamente o exemplo da rede de Petri gerada a partir do mecanismo do tolerância a falhas do MapReduce considerando três componentes, e a Figura 3.6(b) mostra o grafo de alcançabilidade gerado a partir dessa rede.

O vértice inicial do grafo *S0* é a marcação inicial da rede, com uma marca no lugar *Master.ativo* e duas marcas em *Worker.ativo*. O vértice *S1* é criado com o disparo da transição *Master.iniciaAplicação*, representando agora a existência de uma marca em *Worker.executandoMap* e uma em *Worker.ativo*. A partir dessa marcação, representada pelo vértice *S1*, duas transições são possíveis, e consequentemente dois vértices são criados no grafo. O vértice *S2* é criado com o disparo da transição *nada*, e representa a existência de uma marca em *Worker.executandoReduce* e uma em *Worker.ativo*. Seguindo nesse caminho, novamente duas transições podem ser executadas a partir dessa marcação. O vértice *S5* é criado com o disparo da transição *Master.finalizaAplicação*, e representa que existe uma marca em *Worker.fim* e uma em *Worker.ativo*. O vértice *S4* é criado com o disparo da transição *Worker.falha.executandoReduce*, e representa a existência de apenas uma marca em *Worker.executandoReduce*. A partir dessa marcação, o vértice *S6* é criado com o disparo da transição *Master.finalizaAplicação*, e representa que existe apenas uma marca em *Worker.fim*. Voltando ao vértice *S1*, o vértice *S3* é criado com o disparo da transição *Worker.falha.executandoMap*, e representa que existe agora apenas uma marca em *Worker.executandoMap*. O vértice *S4* é novamente alcançado, mas agora pelo disparo da transição *nada*. A transição *Master.finalizaAplicação* pode novamente ser disparada, e encerra o con-



(a) Rede de Petri com três marcas e que representa o mecanismo de tolerância a falhas do MR com três componentes.



(b) Grafo de alcançabilidade da rede de Petri com três marcas, Figura 3.6(a).

Figura 3.6: Exemplo de uma rede de Petri com três marcas e o seu grafo de alcançabilidade.

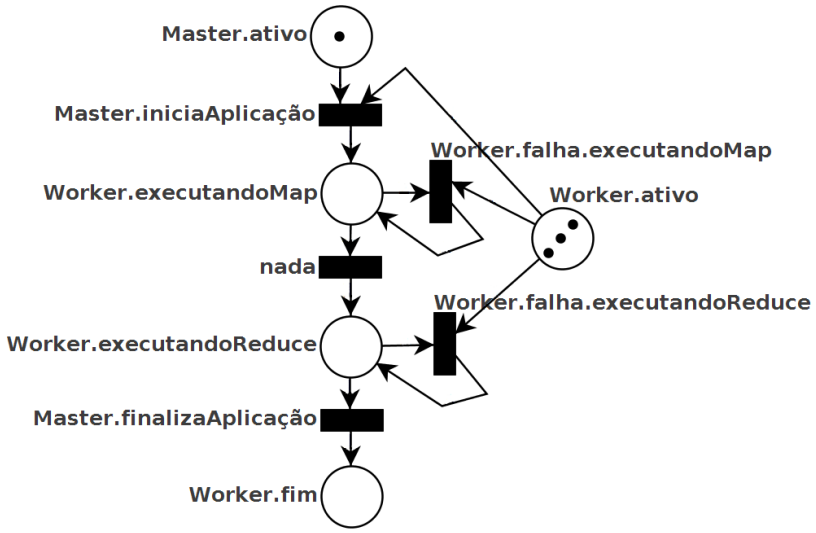
junto de disparos possíveis para essa rede.

O grafo de alcançabilidade obtido é muito semelhante a máquina de estados apresentada na seção anterior, Figura 3.2. Entretanto, o grafo de alcançabilidade representa os componentes e o paralelismo das ações através das marcas que estão distribuídas nos vértices. Além do mais, outras propriedades podem ser utilizadas na RdP, como a identificação por cor dos componentes que estão realizando determinada tarefa, ou a obtenção do número de componentes necessários para a validação de um mecanismo.

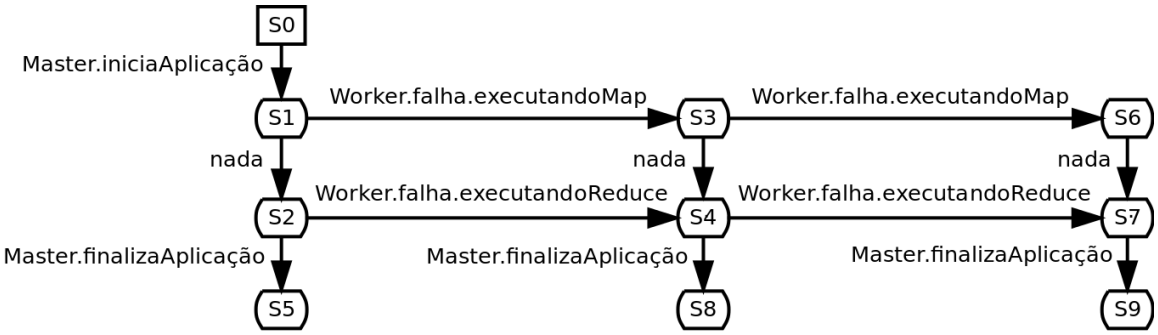
O grafo de alcançabilidade é específico para cada RdP. Qualquer alteração que se faça em uma rede de Petri cria a necessidade de se refazer o seu grafo de alcançabilidade. Assim, todas

as novas possibilidades de disparos das transições são mapeadas.

A Figura 3.7(a) mostra uma rede de Petri com os mesmos lugares e transições da rede de Petri anterior, mostrada na Figura 3.6(a), mas com uma marcação inicial diferente. Foi alterado apenas o número de marcas no lugar *Worker.ativo*, de 2 para 3, representando que foi adicionado mais um *Worker* à execução de um sistema MapReduce. A Figura 3.7(b) mostra o grafo de alcançabilidade gerado a partir desta nova rede de Petri, mostrada na Figura 3.7(a). A construção e a interpretação do grafo seguem a mesma apresentada anteriormente, mas agora o grafo é formado por dez vértices, que representam as dez marcações possíveis por essa rede de Petri.



(a) Rede de Petri com quatro marcas e que representa o mecanismo de tolerância a falhas do MR com quatro componentes.



(b) Grafo de alcançabilidade da rede de Petri com quatro marcas, Figura 3.7(a).

Figura 3.7: Exemplo de uma rede de Petri com quatro marcas e o seu grafo de alcançabilidade.

3.3.2 Processo para Gerar os Casos de Falha

Os casos de falha representativos para o teste de dependabilidade de um sistema MapReduce são gerados a partir do grafo de alcançabilidade da rede de Petri que modela o mecanismo de tolerância a falhas desse sistema. O processo para gerar os casos de falha consiste em percorrer todos os caminhos possíveis do grafo, e mapear cada caminho como um caso de falha.

Conforme Definição 3.1.1, um caso de falha é a quadrupla $\mathcal{F} = (T^{\mathcal{F}}, A^{\mathcal{F}}, R^{\mathcal{F}}, \mathcal{O})$. A lista de testadores $T^{\mathcal{F}}$ é obtida a partir da quantidade de marcas existentes no vértice inicial do grafo. A lista de ações $A^{\mathcal{F}}$ é obtida a partir das arestas que ligam os vértices do grafo. A lista de resultados das ações $R^{\mathcal{F}}$ depende da lista de ações e é instanciada a partir da execução das ações. O oráculo $\mathcal{O}$ é fornecido, conforme descrito na Seção 3.1. Assim, o problema da geração de um caso de falha $\mathcal{F}$, a partir de um grafo de alcançabilidade de uma rede de Petri, recai sobre instanciar a lista de ações $A^{\mathcal{F}}$.

Retomando o grafo de alcançabilidade apresentado na Figura 3.7(b), tem-se que seis caminhos são possíveis a partir do vértice *S0*. A lista de ações de um caso de falha é mapeada a partir de cada caminho do grafo, por exemplo o caminho: *Master.iniciaAplicação*, *Worker.falha.executandoMap*, *nada*, e *Master.finalizaAplicação*.

Da mesma forma que foi informado na modelagem, existem ações que são representadas no grafo de alcançabilidade, mas que não são utilizadas para a geração dos casos de falhas. Essas ações não são mapeadas para a geração dos casos de falha, pois representam o funcionamento normal do sistema MapReduce, ou seja, sem a ocorrência de falhas. As ações que fazem parte de um caso de falha são representadas pelas arestas que possuem identificadores iniciados pela palavra *Master* ou *Worker*.

Conforme apresentado na Definição 3.1.2, uma ação de um caso de falha é a 7-tupla $a_i = (h, D, n, T', I, W, \theta)$. A ordem hierárquica h é obtida a partir da sequência das arestas. O conjunto de ações que devem ser executadas antes, D, é obtido também seguindo a sequência das arestas. O número requerido de execuções n é obtido a partir da marcação dos lugares

nos vértices. O conjunto de testadores que podem executar a ação, T', é obtido a partir dos identificadores das arestas. A instrução I e o gatilho W, também são extraídos dos identificadores das arestas. Entretanto, o limite de tempo θ deve ser atribuído manualmente.

Além das ações extraídas do modelo, as ações de inicialização e finalização do *Master* e do(s) componente(s) *Worker* precisam ser adicionadas para formarem um caso de falha. A Tabela 3.1 apresenta a lista de ações de um exemplo de um caso de falha gerado a partir do grafo.

Tabela 3.1: Lista de ações de um caso de falha gerado a partir do modelo.

	h	D	n	T'	W	θ	I
a_0	1	$\emptyset$	1	$\{t_0\}$		1000	*iniciaMaster*
a_1	2	$\{a_0\}$	3	$\{t_1, t_2, t_3\}$		1000	*iniciaWorker*
a_2	3	$\{a_1\}$	1	$\{t_0\}$		1000000	*enviaAplicação*
a_3	3	$\{a_1\}$	1	$\{t_1, t_2, t_3\}$	$executandoMap$	1000	*falha*
a_4	4	$\{a_2\}$	1	$\{t_0\}$		10000	*validaResultado*
a_5	5	$\{a_1\}$	2	$\{t_1, t_2, t_3\}$		1000	*paraWorker*
a_6	6	$\{a_0\}$	1	$\{t_0\}$		1000	*paraMaster*

Esse caso de falha envolve um conjunto de quatro componentes, $T^{\mathcal{F}} = \{t_0, t_1, t_2, t_3\}$, e sete ações, $A^{\mathcal{F}} = \{a_0, \ldots, a_6\}$. A ação a_0 deve ser executada pelo testador t_0 e inicia o componente *Master*. A ação a_1 deve ser executada pelos testadores $\{t_1, t_2, t_3\}$ e inicia os componentes *Worker*. A ação a_2 deve ser executada pelo t_0 e inicia uma aplicação MapReduce. A ação a_3 deve ser executada paralelamente com a ação a_2 e por um dos testadores $\{t_1, t_2, t_3\}$. A ação a_3 injeta uma falha no *Worker* enquanto ele executa uma tarefa *Map*. A ação a_4 deve ser executada pelo t_0 e valida o resultado. Essa ação foi mapeada a partir da transição *Master.finalizaAplicação*, que representa o final da execução da aplicação e permite a validação do resultado obtido. A ação a_5 finaliza *Worker*, e deve ser executada por apenas dois testadores do conjunto $\{t_0, t_1, t_2\}$, já que a ação a_3 injetou uma falha em um componente. A ação a_6 deve ser executada pelo testador t_0 e finaliza a execução do *Master*.

3.4 Considerações

Para realizar o teste de dependabilidade de um sistema MapReduce é necessário adotar uma abordagem para gerar casos de falha representativos, ao invés de considerar todos os casos possíveis em um ambiente distribuído. Esse capítulo apresentou uma abordagem para gerar casos de falha, que são considerados representativos por serem obtidos a partir do modelo do mecanismo de tolerância a falhas do MapReduce.

A modelagem usando máquina de estados finitos abstrai detalhes do funcionamento do sistema e possibilita uma associação facilitada com as etapas de processamento de um sistema MapReduce. Entretanto, essa abordagem representa os comportamentos dos componentes de forma dependente, necessitando especificar qual o componente deve executar determinada ação.

A modelagem usando redes de Petri possibilitou a apresentação de uma nova abordagem para interpretar os seus componentes. Essa abordagem possibilitou modelar os componentes do MapReduce para que fossem facilmente inseridos e removidos do modelo. Além disso, as ações e os componentes foram modelados de forma independente, possibilitando que uma ação seja executada por qualquer componente habilitado.

Os casos de falha foram então gerados a partir do grafo de alcançabilidade de uma rede de Petri. Esse grafo é formado por todos os disparos possíveis de uma rede de Petri. Cada caminho no grafo de alcançabilidade pode ser mapeado como um caso de falha representativo para o teste de dependabilidade de um sistema MapReduce.

A abordagem apresentada para a geração de casos de falha representativos pode ser utilizada para o teste de dependabilidade de diversos sistemas MapReduce. Os sistemas que possuem um mecanismo de tolerância a falhas equivalente ao modelado neste trabalho, pode ser testado considerando o mesmo conjunto obtido de casos de falha representativos. Além do mais, os sistemas que possuem um mecanismo de tolerância a falhas diferente do modelado, podem ser testados utilizando a mesma abordagem apresentada para a modelagem e a geração de casos de falha representativos para o teste desses sistemas.

A automatização da execução dos casos de falha representativos é o próximo problema objeto deste trabalho. A execução de um caso de falha envolve controlar os componentes do MapReduce que estão distribuídos em diversas máquinas, garantir que as falhas sejam injetadas nos componentes e nos momentos corretos, e validar o comportamento do sistema. Com esse objetivo, o próximo capítulo apresenta a plataforma de teste HadoopTest.

CAPÍTULO 4

AUTOMATIZAÇÃO DA EXECUÇÃO DE CASOS DE FALHA REPRESENTATIVOS

O teste de dependabilidade de um sistema MapReduce envolve a sua validação a partir da execução dos casos de falha representativos de forma experimental e automatizada. A execução experimental dos casos de falha é importante, pois possibilita testar a dependabilidade do sistema considerando as propriedades e características reais do seu ambiente. Já a automatização da execução e validação do comportamento do sistema possibilita que vários casos de falha sejam utilizados para o teste, enquanto que ao mesmo tempo, esconde toda a complexidade do controle do sistema MapReduce e da injeção de falhas.

O sistema responsável por automatizar a execução e a avaliação dos casos de falha é chamado de plataforma de teste. Para testar a dependabilidade de um sistema MapReduce, a plataforma de teste deve controlar a execução dos seus diferentes componentes em diversas máquinas, garantir que as falhas sejam injetadas nos componentes e nos momentos corretos, monitorar o comportamento do sistema, e validar esse comportamento de acordo com o esperado.

Este capítulo introduz o HadoopTest, uma plataforma de testes para implementações baseadas no MapReduce. O HadoopTest possibilita a execução automatizada de casos de falha através de máquinas distribuídas. Na sequência, a Seção 4.2 apresenta detalhes da implementação do HadoopTest.

4.1 HadoopTest: Uma Plataforma de Testes para Sistemas MapReduce

O HadoopTest é uma plataforma proposta neste trabalho com o objetivo de ajudar pesquisadores e profissionais na automatização do controle, monitoramento e validação da execução de casos de falha representativos para o teste de dependabilidade de um sistema MapReduce. Os casos de falha são um conjunto de ações que envolvem a injeção de falhas, e que devem ser executadas pelos diferentes componentes do MapReduce de forma concorrente e distinta. Assim, o HadoopTest possibilita o teste de dependabilidade de um sistema MapReduce, validando o seu comportamento mediante falhas, que são injetadas nos seus diferentes componentes e nos momentos necessários para o teste.

O HadoopTest é uma plataforma baseada no PeerUnit [de Almeida et al., 2008], plataforma de teste funcional para sistemas P2P [Androutsellis-Theotokis and Spinellis, 2004]. O PeerUnit controla a execução de casos de teste distribuídos, que envolvem a execução sequencial de ações que podem ser executadas por conjuntos de componentes distintos. Em [Marynowski et al., 2011] é apresentada a primeira versão do HadoopTest, que estende a plataforma PeerUnit para controlar os diferentes componentes MapReduce e possibilitar a execução de ações em paralelo. Entretanto, a execução dos casos de falha representativos envolve ainda a identificação dinâmica dos testadores que devem executar as ações de injeção de falha, seguindo as etapas de processamento de cada componente MapReduce. A nova versão do HadoopTest com esses requisitos está descrita em [Marynowski and Pimentel, 2013] e neste capítulo.

A arquitetura do HadoopTest segue a apresentada pelo PeerUnit e é mostrada na Figura 4.1. Ela é composta por dois componentes, um *coordenador* e vários *testadores*. O *coordenador* controla a execução dos testadores distribuídos, coordena as ações dos casos de falha, e gera o veredito do caso de falha a partir dos resultados fornecidos pelos testadores. Cada *testador* controla um componente MapReduce: ele recebe as mensagens de coordenação, executa as ações do caso de falha e retorna os seus resultados.

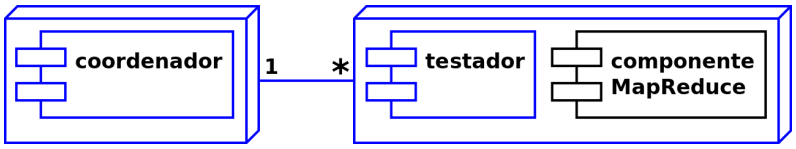


Figura 4.1: Diagrama de componentes do HadoopTest.

A Figura 4.2 mostra a representação do HadoopTest executando um caso de falha sobre uma instância de um sistema MapReduce. Esta execução envolve o controle de um sistema MapReduce executado por quatro componentes, um *Master* e três *Workers*, numerados de 0 a 2. O *coordenador* controla individualmente a execução de quatro testadores, identificados por $\{t0, ..., t3\}$. O testador $t0$ controla o *Master*, e cada outro testador, $\{t1, t2, t3\}$, controla um componente *Worker*. O testador $t0$ submete uma aplicação MapReduce ao *Master* que coordena a sua execução associando as tarefas de *Map* e *Reduce* para os *Workers*. O testador $t2$ injeta uma falha no *Worker1* enquanto ele executa uma tarefa *Map*. Tanto o *Worker0* quanto o *Worker2* realizam a tarefa *Reduce*, e geram o resultado da aplicação nos arquivos *saida0* e *saida1*.

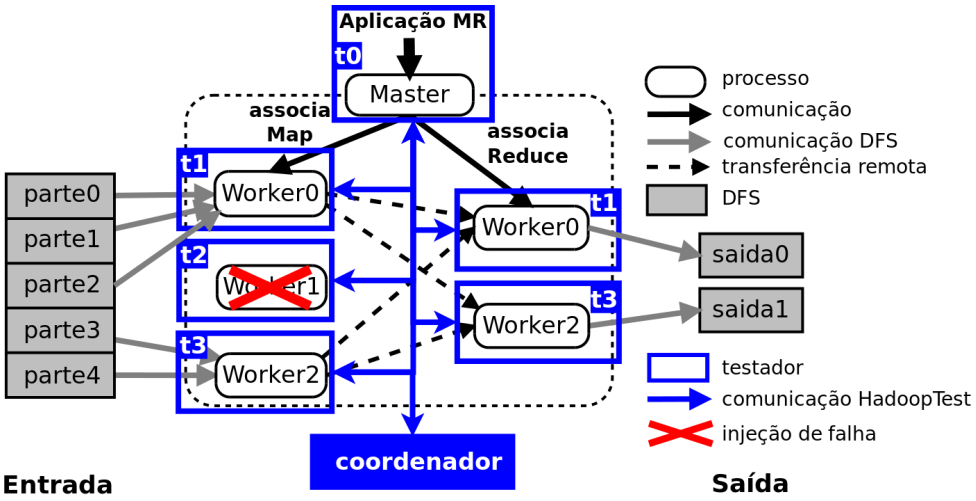


Figura 4.2: Representação do HadoopTest ao executar um caso de falha.

Conforme apresentado na Definição 3.1.1, um caso de falha $\mathcal{F} = (T^{\mathcal{F}}, A^{\mathcal{F}}, R^{\mathcal{F}}, \mathcal{O})$. A lista de testadores $T^{\mathcal{F}}$ e a lista de resultados das ações $R^{\mathcal{F}}$ são produzidas pelo HadoopTest a partir da lista de ações $A^{\mathcal{F}}$. O oráculo implementado é o mesmo disponibilizado pelo PeerUnit, e o seu funcionamento está descrito em mais detalhes na Seção 3.1. Assim, a descrição de um

caso de falha para o HadoopTest pode ser simplificada à especificação da sua lista de ações $A^{\mathcal{F}}$.

A Tabela 4.1 mostra a lista de ações $A^{\mathcal{F}}$ do caso de falha representado na Figura 4.2. O objetivo é validar a execução do sistema MapReduce mediante a falha de um *Worker* enquanto ele executa uma tarefa *Map*.

Tabela 4.1: Exemplo da lista de ações de um caso de falha para o HadoopTest.

	h	D	n	T'	W	θ	I
a_0	1	$\emptyset$	1	$\{t_0\}$		1000	*iniciaMaster*
a_1	2	$\{a_0\}$	3	$\{t_1, t_2, t_3\}$		1000	*iniciaWorker*
a_2	3	$\{a_1\}$	1	$\{t_0\}$		1000000	*enviaAplicacao*
a_3	3	$\{a_1\}$	1	$\{t_1, t_2, t_3\}$	$executandoMap$	1000	*falhaWorker*
a_4	4	$\{a_2\}$	1	$\{t_0\}$		10000	*validaResultado*
a_5	5	$\{a_1\}$	2	$\{t_1, t_2, t_3\}$		1000	*paraWorker*
a_6	6	$\{a_0\}$	1	$\{t_0\}$		1000	*paraMaster*

Esse caso de falha envolve um conjunto de sete ações, $A^{\mathcal{F}} = \{a_0, \ldots, a_6\}$, e deve ser executado como segue. Inicialmente, o testador t_0 executa a ação a_0 para iniciar o componente *Master*. Se a ação a_0 teve como resultado *SUCESSO* e como $D_{a_1} = a_0$, então os outros três testadores, $\{t_1, t_2, t_3\}$ executam a ação a_1 para iniciar os componentes *Worker*. Por outro lado, se a ação a_0 teve como resultado *FALHO*, então a ação a_1 termina e também recebe *FALHO* como resultado. Isso acontece com qualquer ação que tem uma relação de dependência com uma ação com resultado *FALHO*, de forma recursiva. Sem ações falhas, o processo continua e o testador t_0 executa a ação a_2 que submete uma aplicação MapReduce ao *Master*. Paralelamente, todos os testadores do conjunto $\{t_1, t_2, t_3\}$ recebem a ação a_3, que injeta uma falha de interrupção (*crash*) no *Worker* em execução em cada testador. Entretanto, apenas o primeiro testador que executar uma tarefa *Map* sofre a falha, uma vez que $n_{a_3} = 1$ e W_{a_3}=*executandoMap()*. O testador t_0 executa a ação a_4 na qual ocorre a validação do comportamento do sistema através da função *validaResultado()*. Esta função verifica se o resultado obtido é igual ao esperado, e em caso afirmativo o resultado da ação é *SUCESSO*, caso contrário, o resultado é *FRACASSO*. Agora, se t_0 não é capaz de obter qualquer resultado do sistema dentro do tempo limite da ação a_4, que é $\theta = 1000$, então o resultado é *INCON-*

CLUSIVO. Como um testador injetou uma falha no seu componente, a ação a_5, que finaliza um componente *Worker*, é executada apenas por dois testadores do conjunto t_0, t_1, t_2. E o testador t_0 executa a última ação a_6 que é a finalização do componente *Master*. O veredito do caso de falha será *PASSOU* se todas as ações obtiverem *SUCESSO* como resultado, será *FALHOU* se qualquer ação tiver como resultado *FRACASSO*, ou será *INCONCLUSIVO* se não puder fazer nenhuma das afirmações anteriores.

4.1.1 A Coordenação de um Caso de Falha

A execução de um caso de falha consiste na coordenação e controle dos testadores para que executem as ações de forma distribuída, paralela e sincronizada. O Algoritmo 4 apresenta os principais passos realizados para coordenar os testadores na execução das ações de um caso de falha $\mathcal{F}$. Para cada nível hierárquico h, especificado no conjunto de ações $A^{\mathcal{F}}$, o coordenador executa três passos principais: (1) envia mensagens para os testadores para que executem as ações em paralelo; (2) recebe os resultados locais das execuções das ações pelos testadores; e (3) atribui o resultado final das ações, $R^{\mathcal{F}}$, com base nos resultados locais obtidos pelos testadores. Após a execução das ações de todos os níveis, o oráculo $\mathcal{O}$ atribui o veredito do caso de falha com base nos resultados das ações.

Algoritmo 4: Coordenação de um caso de falha

Entrada: $\mathcal{F}$, um caso de falha; $\mathcal{M}$, uma função de mapeamento entre o conjunto $A^{\mathcal{F}}$ e os níveis hierárquicos de suas ações.
Dados: n_r, o número de resultados de ações com sucesso; R_l, um conjunto dos resultados locais dos testadores ao executar suas ações.
Saída: um veredito.
Início
  Para cada $h \in \mathcal{M}(A^{\mathcal{F}})$ **faça**
    $n_r \leftarrow$EnviaMensagens$(\mathcal{M}^{-1}(h), R^{\mathcal{F}})$
    $R_l \leftarrow$RecebeRespostas$(\mathcal{M}^{-1}(h), n_r)$
    $R^{\mathcal{F}} \leftarrow$ProcessaResultados$(\mathcal{M}^{-1}(h), R_l)$
  fim
  Retorna $\mathcal{O}(R^{\mathcal{F}})$
fim

O Algoritmo 5 apresenta a função *EnviaMensagens*, que envia as mensagens de execução

das ações para os testadores especificados. Esta função recebe um conjunto de ações com o mesmo nível hierárquico, verifica as suas relações de dependência, e se não tiver problemas, envia as mensagens de controle para os testadores especificados executarem as ações. Ao final, a função *EnviaMensagens* retorna o número de resultados das ações que devem obter sucesso na sua execução.

Algoritmo 5: EnviaMensagens

Entrada: A', um conjunto de ações de um mesmo nível hierárquico; $R^{\mathcal{F}}$, o conjunto de resultados das ações.
Saída: n_r, o número de resultados de ações com sucesso.
Início
  $n_r \leftarrow 0$
  Para cada $a_i \in A'$ **faça**
    Se $R^{\mathcal{F}}[a_j] = SUCESSO, \forall a_j \in D_{a_i}$ **então**
      Envia uma mensagem para a execução de a_i para todos $t \in T'_{a_i}$
      $n_r \leftarrow n_r + n_{a_i}$
    fim
    senão
      $R^{\mathcal{F}}[a_i] \leftarrow FRACASSO$
    fim
  fim
  Retorna n_r
fim

A entrada para a função $EnviaMensagens$ é o conjunto de ações que devem ser executadas em paralelo A', e o conjunto de resultados das ações $R^{\mathcal{F}}$. Para cada ação a_i pertencente ao conjunto de ações A', é verificado se todas as ações anteriores, citadas na sua relação de dependência D_{a_i}, executaram com sucesso. Nesse caso, as mensagens de execução da ação a_i são enviadas para os testadores especificados no seu conjunto T'_{a_i}, e o número de respostas necessárias n_r é registrado. Caso contrário, ou seja, a ação a_i não pode executar pois as ações anteriores não executaram com sucesso, o resultado da ação a_i é *FRACASSO*. Esse passo é finalizado retornando o número de respostas necessárias para as ações em execução pelos testadores.

O Algoritmo 6 mostra a função $RecebeRespostas$ que consiste no segundo passo da

coordenação da execução de um caso de falha. Esta função recebe os resultados das ações enquanto o número requirido de execuções com sucesso e o tempo limite não é atingido. A função *RecebeRespostas* verifica se o resultado pode ser considerado, conferindo o número requirido de execuções com sucesso da ação relacionada, e retorna o conjunto de resultados locais dos testadores.

Algoritmo 6: RecebeRespostas

Entrada: A', um conjunto de ações; n_r, o número de resultados das ações com sucesso.
Saída: R_l, um conjunto dos resultados locais dos testadores ao executar as suas ações.
Início
  Enquanto $(n_r > 0) \wedge (clock < \theta_{a_i}, \forall a_i \in A')$ **faça**
    Recebe resposta r de t e identifica a ação a_i por ele executada
    Se $(n_{a_i} = |T'_{a_i}|)$ **então**
      $n_r \leftarrow n_r - 1$
      $R_l[t] \leftarrow r$
    fim
    senão
      Se $(n_{a_i} > 0)$ **então**
        Se $(r = SUCESSO)$ **então**
          $n_r \leftarrow n_r - 1$
          $n_{a_i} \leftarrow n_{a_i} - 1$
          $R_l[t] \leftarrow r$
        fim
      fim
    fim
  fim
  Retorna R_l
fim

A função *RecebeRespostas* é executada enquanto o número de respostas necessárias ou o maior tempo limite das ações em execução não sejam atingidos. O coordenador identifica qual a ação a_i refere-se a resposta r e a classifica de acordo com o número de respostas necessárias. Caso o número de respostas necessárias n_{a_i} for menor que o tamanho de T'_{a_i}, o resultado é verificado e se for *SUCESSO* o n_{a_i} é decrementado, caso contrário a resposta é desprezada. Caso o número de respostas for igual ao tamanho de T'_{a_i}, o resultado local dos testadores é armazenado no conjunto de resultados locais dos testadores R_t, que é retornado para posterior validação.

Por fim, o Algoritmo 7 mostra a função $ProcessaResultado$ que é o último passo para a coordenação da execução das ações em paralelo. Essa função processa os resultados locais das ações executadas em paralelo e atribui um resultado único para cada ação. O resultado de cada ação obtido forma o conjunto de resultados das ações $R^{\mathcal{F}}$, que é retornado para o processamento do caso de falha. Esse conjunto é utilizado na verificação da relação de dependência das ações e para a obtenção do veredito do caso de falha, realizada pelo oráculo.

Algoritmo 7: ProcessaResultados

Entrada: A', um conjunto de ações; R_l, um conjunto dos resultados locais dos testadores ao executar suas ações.
Saída: $R^{\mathcal{F}}$, o conjunto de resultados das ações.
Início
  Para cada $a_i \in A'$ **faça**
    Se $n_{a_i} = 0$ **então**
      $R^{\mathcal{F}}[a_i] \leftarrow SUCESSO$
    fim
    Se $R_l[t] = SUCESSO, \forall t \in T'_{a_i}$ **então**
      $R^{\mathcal{F}}[a_i] \leftarrow SUCESSO$
    fim
    senão
      Se $\exists r \in R_l[t] : r = FRACASSO, \forall t \in T'_{a_i}$ **então**
        $R^{\mathcal{F}}[a_i] \leftarrow FRACASSO$
      fim
      senão
        $R^{\mathcal{F}}[a_i] \leftarrow TIMEOUT$
      fim
    fim
  fim
  Retorna $R^{\mathcal{F}}$
fim

A função *ProcessaResultados* inicia verificando se todos os resultados locais dos testadores são *SUCESSO*. Caso afirmativo, então o resultado da ação também é *SUCESSO*. Caso exista algum resultado local dos testadores com *FRACASSO*, então o resultado da ação também é *FRACASSO*. Agora, se a função não obteve resposta de qualquer testador no tempo limite, então o resultado da ação é *TIMEOUT*.

4.1.2 A Execução das Ações pelos Testadores

Um testador segue quatro passos para executar uma ação de um caso de falha. Primeiro, o testador recebe a mensagem de execução de uma determinada ação. Segundo, ele aguarda um estado determinado do componente para executar essa ação. Terceiro, o testador verifica se ele ainda pode executar a ação. E quarto, ele executa a instrução ou comando especificado na ação.

O Algoritmo 8 apresenta os detalhes dos quatro passos relatados. O processamento se inicia com o recebimento de uma mensagem para a execução da ação a_i. Se o gatilho W_{a_i} é definido então o testador aguarda a sua execução. Após a execução do gatilho, ou se o gatilho não foi definido, o testador verifica se o número de sucesso nas respostas da ação n_{a_i} é maior que zero. Se for maior que zero, então ele executa a instrução I_{a_i}, e retorna o resultado dessa execução. Caso contrário, o testador retorna *FRACASSO*, informando ao coordenador do caso de falha que não pode executar a ação.

Algoritmo 8: Executa uma ação de um caso de falha

Dados: a_i, uma ação de um caso de falha.
Saída: Um resultado da execução.
Início
  $a_i \leftarrow$ Recebe mensagem para a execução de uma ação
  Se $W_{a_i} \neq NULL$ **então**
    Execute W_{a_i}
  fim
  Se $n_{a_i} > 0$ **então**
    Retorna Execute I_{a_i}
  fim
  Retorna $FRACASSO$
fim

4.2 Implementação do HadoopTest

A plataforma de teste HadoopTest foi desenvolvida com o objetivo de auxiliar desenvolvedores e pesquisadores na realização teste de dependabilidade de um sistema MapReduce. O HadoopTest está disponível para download a partir do endereço:

http://www.inf.ufpr.br/jeugenio/HadoopTest.tar.gz

A implementação do HadoopTest provê suporte para testar o Hadoop [Apache, 2013c], principal implementação de código aberto do MapReduce, mas pode ser adaptada para testar outros sistemas MapReduce, ou ainda outros sistemas distribuídos. A interface com o sistema a ser testado pode ser implementada via uma classe abstrata ou os *scripts* de manipulação do sistema, como executar o sistema, injetar falhas e identificar o momento em que a falha deve ser injetada.

O Hadoop é implementado em Java e sua arquitetura é composta principalmente por dois componentes: o *JobTracker* e o *TaskTracker*. O *JobTracker* implementa o componente *Master* do MapReduce; e o *TaskTracker* implementa o componente *Worker*. Juntamente com o Hadoop é disponibilizado o HDFS (*Hadoop File System*) que é o sistema de arquivos distribuído utilizado pelo Hadoop para armazenar o dado de entrada e o resultado do processamento [Apache, 2012]. O HDFS tem uma arquitetura mestre/escravo, assim como o Hadoop. O componente mestre do HDFS é único e chamado de *NameNode*, ele gerencia o espaço de nomes (*namespace*) do sistema de arquivo e o controle de acesso pelos usuários. Os componentes escravos são chamados de *DataNode* e gerancia o armazenamento do dado distribuído em cada máquina em que, normalmente, ele é executado.

A Figura 4.3 apresenta as principais classes desenvolvidas para implementar a interface entre o Hadoop e o HadoopTest. A classe *AbstractMR* implementa os métodos utilizados no desenvolvimento de um caso de falha, apresentando os componentes do Hadoop e HDFS, e, principalmente, os métodos utilizados nas ações de um caso de falha. As implementações dos componentes são realizadas nas classes *HadoopNameNodeWrapper*, *HadoopDataNodeWrapper*, *HadoopJobTrackerWrapper*, *HadoopTaskTrackerWrapper*.

Conforme citado anteriormente, o HadoopTest é uma extensão do PeerUnit [de Almeida et al., 2010b]. Assim, o HadoopTest também é implementado em Java e faz uso intensivo da reflexão dinâmica e de anotações, usando estas características para selecionar e executar as ações que compõem um caso de teste. Detalhes da implementação e descrição de um caso de falha é descrito na Seção 4.2.2.

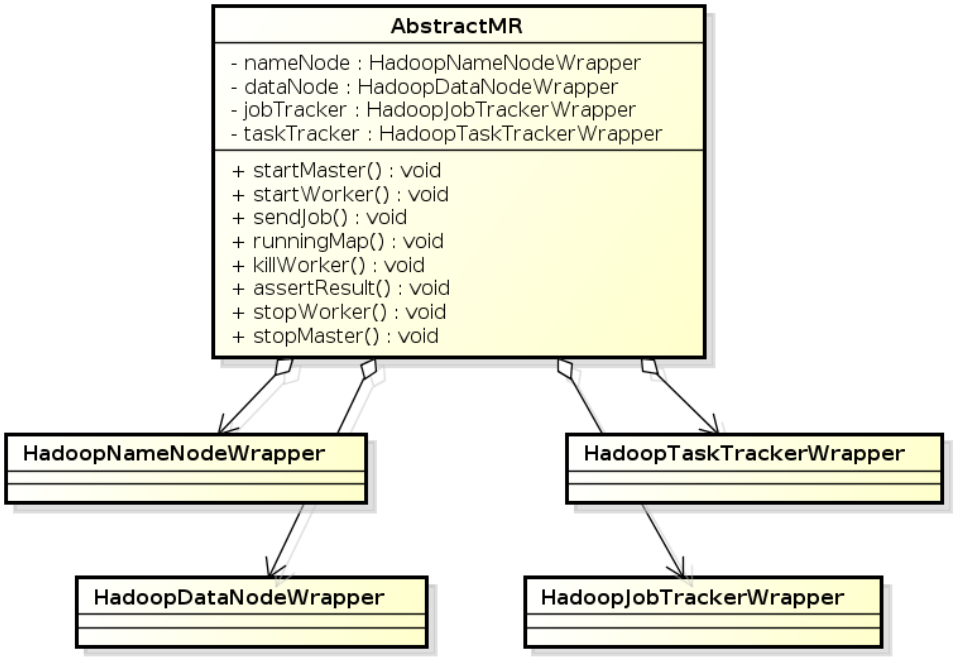


Figura 4.3: Classes da interface entre o Hadoop e HadoopTest.

4.2.1 Principais Componentes e Classes

Os principais componentes do HadoopTest são os mesmos do PeerUnit, sendo estes o Coordenador e o Testador. Como mencionado, o Coordenador é responsável por conduzir a execução do caso de falha através dos vários Testadores distribuídos. A comunicação é feita usando *Remote Method Invocation* (RMI), que possibilita que métodos de objetos remotos Java sejam invocados a partir de máquinas virtuais Java (*Java Virtual Machines* - JVM), possivelmente em máquinas físicas diferentes.

As principais interfaces dos componentes e tipos de dados do HadoopTest permanecem as mesmas do PeerUnit. A interface *Coordinator* representa o Coordenador, *Tester* representa cada Testador, *MethodDescription* representa cada ação de um caso de falha, e *Verdicts* representa os vereditos possíveis: *PASS* (PASSOU), *FAIL* (FALHOU), *INCONCLUSIVE* (INCONCLUSIVO), e *ERROR* (ERRO).

Cada ação de um caso de falha é uma instância da classe *MethodDescription*. Essa classe foi alterada adicionando e manipulando os novos atributos: *answers*, *when*, e *depend*. Esses atributos foram adicionados para refletir a definição de uma ação de um caso de falha con-

forme Definição 3.1.2. O atributo *answers* é o número de respostas necessárias, *when* é o comando que deve ser executado para habilitar a execução da ação, e *depend* é o conjunto de identificadores de ações que devem ser executadas com sucesso antes da execução da ação. Esses atributos serão vistos em mais detalhes na Seção 4.2.2.

O HadoopTest estende a implementação dos componentes Coordenador e Testador para possibilitar a execução de um caso de falha. A Figura 4.4 apresenta o diagrama de classe simplificado da implementação desses componentes. Principalmente, a extensão ocorreu envolvendo a classe *CoordinatorImpl*, que implementa a interface *Coordinator*, e a classe *CoordinationStrategy*, que é uma interface para uma estratégia de coordenação. O HadoopTest implementa quatro estratégias de coordenação: *SequencialStrategy*, *DependencyStrategy*, *HierarchicalStrategy*, e *GlobalStrategy*. A estratégia *SequencialStrategy* executa as ações sequencialmente aguardando a execução de cada ação por todos testadores referenciados. A estratégia *DependencyStrategy* executa as ações verificando se as ações listadas na relação de dependência foram executadas com sucesso. A estratégia *HierarchicalStrategy* executa paralelamente as ações que possuem o mesmo nível hierárquico. A estratégia *GlobalStrategy* executa paralelamente as ações que possuem o mesmo nível hierárquico e leva em consideração a relação de dependência.

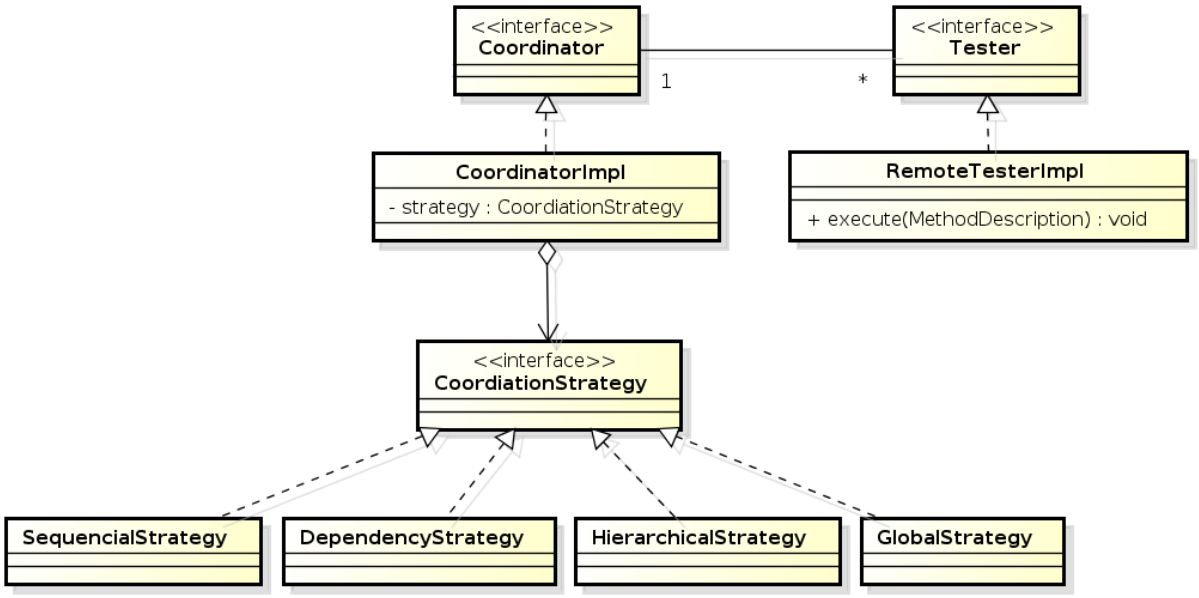


Figura 4.4: Classes principais do Coordenador e Testador no HadoopTest.

A alteração envolvendo o componente Testador foi realizada na classe *RemoteTesterImpl*, que implementa a classe de interface *Tester*. O método *execute(MethodDescription)* foi alterado para realizar o procedimento de execução das ações pelos testadores, conforme descrito no Algoritmo 8.

A Figura 4.5 apresenta o diagrama de sequência da execução de uma ação por um Testador. A ação utilizada é componente da lista de ações apresentada anteriormente, na Tabela 4.1. Em um determinado momento durante a execução do caso de falha, o Coordenador requisita a execução da ação a_3 ao Testador t_1. O Testador aguarda o estado do componente *Worker0* executando o comando *executandoMap()*. Posteriormente, o Testador requisita ao Coordenador o número de respostas necessárias, executando o método *getAnswers()*, e caso ainda não seja suficiente (maior que zero) executa o comando *falhaWorker()*, que injeta uma falha no *Worker0*.

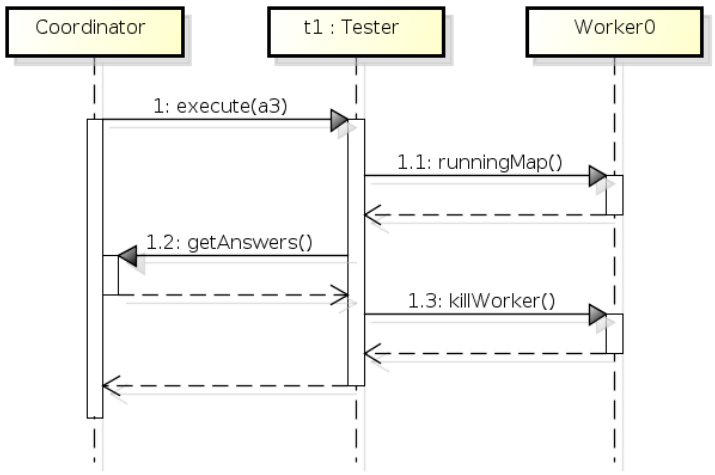


Figura 4.5: Diagrama de sequência da execução de uma ação por um Testador.

4.2.2 Escrevendo um Caso de Falha

Um caso de falha $\mathcal{F}$ é descrito e interpretado pelo HadoopTest como uma classe Java. Conforme citado na Seção 3.1, a lista de testadores $T^{\mathcal{F}}$, a lista de resultados das ações $R^{\mathcal{F}}$ e o oráculo são disponibilizados pelo HadoopTest. A descrição de um caso de falha é então simplificada para a descrição da lista de ações do caso de falha, que são os métodos da classe

Java que possuem a anotação *@TestStep*.

As anotações são metadados onde são definidos os elementos de uma ação que são utilizados durante a coordenação e execução de cada ação. O PeerUnit disponibiliza três anotações para métodos: *@BeforeClass*, que indica que o método é o primeiro a ser executado; *@AfterClass*, que indica que o método é o último a ser executado; e *@TestStep*, que indica que o método é uma ação de um caso de teste. As anotações *@BeforeClass* e *@AfterClass* normalmente são utilizadas com instruções comuns e necessárias para todos os testadores, e que envolvem a inicialização e a finalização de um caso de teste independente das suas ações. Como exemplo, essas anotações são usadas para deixar cada componente do sistema preparado para a execução do caso de teste, e para retornar o sistema ao mesmo estado que estava antes da execução do teste.

A inicialização e finalização de um caso de falha para sistemas MapReduce, envolve a execução de ações diferentes para os seus componentes e, por consequência, para seus testadores. Assim, foi utilizada apenas a notação *@TestStep* para mapear as ações necessárias para a execução de um caso de falha. Todo método que possui essa notação é interpretado como uma ação de um caso de falha. O PeerUnit disponibiliza três atributos para essa notação:

— *order* - é um número inteiro positivo que define a ordem da ação;

— *range* - é um conjunto de identificadores dos testadores que devem executar a ação;

— *timeout* - é um inteiro que define o intervalo de tempo para que a ação seja finalizada.

O HadoopTest estende esses atributos para refletir os elementos de uma ação de um caso de falha, conforme descrito na Definição 3.1.2. Assim, os atributos disponíveis para a notação *@TestStep*, passam a ser:

— *order* - é o elemento h_{a_i}, e que é um número inteiro positivo, que define o nível hierárquico da ação;

— *depend* - é o elemento D_{a_i}, que é uma sequência de caracteres opcional, composta por um conjunto de identificadores de ações (métodos com a notação *@TestStep*), e que devem ser executadas com sucesso antes da execução desta ação;

— *answers* - é o elemento n_{a_i}, que é um número inteiro positivo e opcional, que define o número de respostas necessárias;

— *range* - é o elemento T'_{a_i}, que é uma sequência de caracteres, composta por um conjunto de identificadores dos testadores (inteiros positivos) separados por virgula, ou uma faixa de números inteiros positivos (por exemplo, "1-3"), ou ainda um asterisco para que todos os testadores executem este método;

— *when* - é o elemento W_{a_i}, que é uma sequência de caracteres, composta por um comando que deve ser executado para habilitar a execução da ação;

— *timeout* - é o elemento θ_{a_i}, e é um número inteiro positivo interpretado em milissegundos, e é o intervalo de tempo para que a ação seja finalizada.

A Listagem 4.1 apresenta uma parte inicial do código fonte de uma classe Java chamada *Exemplo1*. Essa classe descreve a lista de ações de um caso de falha representativo, previamente apresentado na Tabela 4.1. A classe *Exemplo1* foi dividida para facilitar o entendimento e a visualização. O objetivo da classe é implementar um caso de falha que teste um sistema MapReduce enquanto executa uma aplicação, e uma falha é inserida em um *Worker* que executa uma tarefa *Map*.

```
1 public class Exemplo1 extends AbstractMR{
2   @TestStep(order=1, depend="", answers=1, range="0", when="",
       timeout=100)
3   public void a0() {
4     iniciaMaster();
5   }
```

```
6    @TestStep(order=2, depend="a0", answers=3, range="1-3", when="",
         timeout=1000)
7    public void a1() {
8      iniciaWorker();
9    }
10   ...
11 }
```

Listagem 4.1: Primeira parte da classe *Exemplo1*.

A classe *Exemplo1* é uma subclasse de *AbstractMR*, que implementa a biblioteca do sistema MapReduce e provê os métodos que abstraem a complexidade de manipulá-lo, como o método *iniciaWorker()*, que inicia um componente *Worker*. O primeiro método da classe é o *a0* e descreve uma ação do caso de falha, uma vez que possui a notação *@TestStep*. O objetivo desse método é iniciar o componente *Master*, e os atributos da sua notação informam que o método *a0*:

— deve ser executado inicialmente (*order=1*);

— não possui relação de dependência (*depend=“”*);

— deve ter uma resposta com sucesso (*answers=1*);

— deve ser executado pelo testador t_0 (*range=0*);

— não possui gatilho (*when=“”*); e

— deve ser executado até o tempo 100 (*timeout=100*).

O segundo método da classe *Exemplo1* é o *a1*. O objetivo desse método é iniciar o componente *Worker*, e os atributos da sua notação informam que o método *a1*:

— deve ser executado após o método anterior (*order=2*);

— deve ser executado apenas se a ação *a0* for executada com sucesso (*depend=“a0”*);

— deve ter três respostas com sucesso (*answers=3*);

— deve ser executado pelos testadores t_1, t_2, t_3 (*range="1-3"*);

— não possui gatilho (*when=""*); e

— deve ser executado até o tempo 1000 (*timeout=1000*).

A Listagem 4.2 apresenta a segunda parte da classe *Exemplo1*. Nessa parte são descritas as ações de submissão da aplicação e de injeção da falha, que devem ser executadas em paralelo. O método *a2* envia a aplicação MapReduce e deve ser executado pelo testador t_0, que controla o *Master*. O método *a3* injeta a falha no *Worker* e deve ser executado pelo primeiro testador que executar o gatilho *executandoMap*. Ambos os métodos *a2* e *a3* devem ser executados em paralelo, pois possuem o mesmo nível hierárquico (*order=3*), e após a execução correta do método *a1* (*depend=a1*).

```
1  ...
2  @TestStep(order=3, depend="a1", answers=1, range="0", when="",
       timeout=900000)
3  public void a2() {
4    enviaAplicacao();
5  }
6  @TestStep(order=3, depend="a1", answers=1, range="1-3",
7      when="executandoMap", timeout=1000)
8  public void a3() {
9    falhaWorker();
10 }
11 ...
```

Listagem 4.2: Segunda parte da classe *Exemplo1*.

A Listagem 4.3 apresenta a terceira e última parte da classe *Exemplo1*. Agora, são descritas a ação de validação do resultado da aplicação e as ações de finalização dos componentes

MapReduce. O método *a4* valida o resultado da aplicação MapReduce, e deve ser executado após a execução correta do método *a2* (*depend=a2*), e deve ser executado pelo testador t_0, que controla o *Master*. O método *a5* finaliza o componente *Worker* e deve ser executado com sucesso por dois testadores do conjunto t_1, t_2, t_3 (*answers=2 e range="1-3"*. O método *a6* finaliza o componente *Master* e também deve ser executado pelo testador t_0.

```
1    ...
2    @TestStep(order=4, depend="a2", answers=1, range="0", when="",
        timeout=10000)
3    public void a4() {
4      validaResultado();
5    }
6    @TestStep(order=5, depend="a1", answers=2, range="1-3", when="",
         timeout=1000)
7    public void a5() {
8      paraWorker();
9    }
10   @TestStep(order=6, depend="a0", answers=1, range="0", when="",
        timeout=1000)
11   public void a6() {
12     paraMaster();
13   }
14 }
```

Listagem 4.3: Terceira parte da classe *Exemplo1*.

4.2.3 Executando um Caso de Falha

Para executar um caso de falha no HadoopTest é necessário que o Hadoop esteja configurado para ser executado no conjunto de máquinas a ser utilizado, e parametrizar adequada-

mente os arquivos de controle do HadoopTest, o *peerunit.properties* e o *hadoop.properties*. O próximo passo é executar o *coordenador* do HadoopTest na mesma máquina que o *JobTracker* do Hadoop foi configurado. Em seguida, executar o *testador* do HadoopTest em cada outra máquina e aguardar a finalização da execução do caso de falha. O veredito do caso de falha é apresentado pelo *coordenador* na saída padrão e no arquivo de registro (*log*) do HadoopTest.

4.3 Considerações

A plataforma de teste HadoopTest possibilita a automatização da execução de casos de falha representativos para o teste de dependabilidade de sistemas MapReduce. Esse capítulo apresentou como o HadoopTest controla a execução dos diferentes componentes do MapReduce, como ele garante que as falhas são injetadas nos componentes e nos momentos especificados, e como ele valida o comportamento do sistema de acordo com o esperado.

O HadoopTest apresenta uma arquitetura clássica composta por um coordenador e vários testadores. Cada testador controla um componente do MapReduce e executa as ações de um caso de falha. O coordenador orquestra a execução de um caso de falha e garante que as falhas sejam injetadas nos componentes e nos momentos especificados. A validação do comportamento do sistema ocorre pela verificação de que todas as ações requeridas foram executadas corretamente, e que os resultados obtidos são iguais aos esperados.

A descrição e a execução de um caso de falha ocorrem de uma forma simplificada no HadoopTest. Um caso de falha é descrito apenas pela sua lista de ações, da qual o HadoopTest extrai o conjunto de testadores e gera a lista de resultados das ações. Além do mais, a execução de um caso de falha ocorre apenas iniciando os componentes do HadoopTest, o coordenador e os testadores.

O HadoopTest possibilita o teste de outros sistemas distribuídos de forma facilitada. Para testar outro sistema, é necessário criar uma classe abstrata que implemente a biblioteca do sistema e que disponibilize os métodos que correspondam com as instruções e comandos utilizados nos casos de falha, como um método para iniciar um componente do sistema ou

injetar uma falha.

O próximo capítulo deste trabalho apresenta resultados experimentais obtidos através do uso do HadoopTest para realizar o teste de dependabilidade do sistema Hadoop. A abordagem para a geração dos casos de falha representativos e o HadoopTest são avaliados de acordo com critérios conhecidos para a avaliação de técnicas de injeção de falhas.

CAPÍTULO 5

RESULTADOS EXPERIMENTAIS

Este capítulo apresenta os resultados experimentais para o teste de dependabilidade do Hadoop [Apache, 2013c], principal implementação de código aberto do MapReduce [Dean and Ghemawat, 2004]. O teste de dependabilidade tem o objetivo de validar o comportamento de um sistema mediante falhas, e encontrar erros considerando o seu mecanismo de tolerância a falhas. O teste é realizado utilizando duas contribuições desse trabalho: uma abordagem para a geração de casos de falha representativos, e uma plataforma de teste para a automatização da execução de casos de falha.

A execução de alguns casos de falha resultam na identificação de erros no Hadoop e comprovam que os casos de falha gerados a partir do modelo são representativos para o teste do sistema Hadoop. Entretanto, a validação da plataforma de teste requer uma avaliação das suas propriedades, a partir da execução de diversos casos de falha. Assim, este capítulo apresenta os resultados da execução de diversos casos de falha representativos. Os resultados são agrupados conforme algumas propriedades desejáveis para uma plataforma de teste. Essas propriedades foram baseadas na caracterização de técnicas de injeção de falhas proposta por *Arlat et al.* em [Arlat et al., 2003].

Oito seções compõem este capítulo, que apresenta inicialmente o ambiente computacional utilizado e o sistema MapReduce testado. As seções seguintes apresentam as propriedades avaliadas, que são a controlabilidade, a medição temporal, a não-intrusividade, a repetibilidade e a reprodutibilidade, e, por fim, a eficácia.

5.1 Ambiente de Execução

Os experimentos foram realizados utilizando o Grid'5000 [Grid5000, 2013], projeto francês para disponibilizar um grande conjunto de máquinas em uma plataforma altamente reconfigu-

rável, controlável e monitorável. Foram utilizadas duzentas máquinas que estavam conectadas por uma rede de 1Gbps e possuíam uma configuração similar, composta de dois processadores Intel Xeon 2.6GHz dual-core, 8 GB de memória RAM, disco rígidos SATA de 250 GB, e sistema operacional Debian GNU/Linux 5.0.

5.2 Hadoop e as Aplicações MapReduce de Exemplo

A implementação original do MapReduce é uma solução proprietária do Google e não é disponibilizada para utilização pela comunidade. Existem outras implementações de código aberto, tais como Hadoop [Apache, 2013c], Phoenix [Ranger et al., 2007], Skynet [Pisoni, 2012] e Disco [Papadimitriou and Sun, 2008]. O Hadoop é a implementação mais popular entre elas, sendo mantida pela fundação Apache[1] e utilizada por um grande número de empresas, tais como: Yahoo, Facebook, Ebay, Linkdln e Quantcast [Apache, 2013d].

O objetivo deste trabalho é realizar o teste de dependabilidade de um sistema MapReduce, em particular da sua implementação Hadoop. Os casos de falha são os gerados segundo o mecanismo de tolerância a falha do MapReduce. A versão do Hadoop utilizada para a realização dos experimentos aqui descritos foi a 0.22.0.

O Hadoop disponibiliza aos seus usuários diversas aplicações MapReduce já implementadas. O *PiEstimator* é uma aplicação disponibilizada que calcula o valor do π usando o método de Monte Carlo. Esse método considera um círculo inscrito exatamente dentro de um quadrado com o tamanho do lado igual à 1. A função *Map* cria pontos aleatórios dentro do quadrado, e conta os pontos localizados dentro e fora do círculo. A função *Reduce* acumula os pontos dentro (I) e fora (O) contados pela função *Map*. O valor estimado de π é obtido por $4*(I/T)$, onde $T = I + O$.

O *WordCount* também é uma aplicação disponibilizada pelo Hadoop. Essa aplicação calcula a quantidade de palavras distintas existente em arquivos de texto. A função *Map* recebe como entrada o nome de um arquivo e o seu conteúdo, e para cada palavra encontrada, a função gera como resultado um par composto pela palavra e pelo número 1. A função *Reduce*

[1] http://www.apache.org

acumula os valores encontrados para cada palavra, e gera como resultado um par composto pela palavra e o número de ocorrências resultante.

5.3 Controlabilidade

A controlabilidade é uma propriedade da plataforma de teste que denota a sua habilidade em controlar *onde* e *quando* as falhas são injetadas. Para o teste de dependabilidade de um sistema MapReduce a plataforma deve controlar individualmente todos os componentes distribuídos do MapReduce, de forma a determinar em qual deles (*onde*) e em qual estado este componente se encontra (*quando*). Apenas durante a execução é que essas informações são disponibilizadas, e em cada execução pode ocorrer uma nova configuração dessas informações. Assim, a plataforma deve controlar individualmente a execução de cada componente distribuído do Hadoop, e possibilitar que a falha seja injetada somente no componente que esteja no estado especificado.

Para demonstrar a alta controlabilidade do HadoopTest foram utilizados três casos de falha. Esses casos de falha foram ainda agrupados para facilitar a caracterização do controle dinâmico e estendido possibilitado pelo HadopTest, e que são descritos a seguir.

5.3.1 Controle Dinâmico

O controle dinâmico do HadoopTest é comprovado por sua habilidade em controlar os componentes do Hadoop durante cada execução. Para comprovar essa característica, um mesmo caso de falha foi executado em dois conjuntos distintos de máquinas. O objetivo é demonstrar que o HadoopTest controla os componentes do Hadoop de forma diferenciada em cada execução.

O caso de falha utilizado tem como objetivo validar a execução do Hadoop enquanto dois de seus componentes falham: um enquanto executa uma tarefa *Map*, e o outro enquanto executa uma tarefa *Reduce*. A Tabela 5.1 mostra as ações que compõem esse caso de falha.

As ações $\{a_0, a_1, ..., a_7\}$ são executadas nessa sequência, iniciando pela ação a_0. Entre-

Tabela 5.1: Caso de falha para a validação do controle dinâmico do HadoopTest.

	h	D	n	T'	W	θ	I
a_0	1	$\emptyset$	1	$\{t_0\}$		9000	$iniciaJobTracker$
a_1	2	$\{a_0\}$	3	$\{t_1, t_2, t_3\}$		1000	$iniciaTaskTracker$
a_2	3	$\{a_1\}$	1	$\{t_0\}$		900000	$submeteWordCount$
a_3	3	$\{a_1\}$	1	$\{t_1, t_2, t_3\}$	$executandoMap$	1000	$falhaTaskTracker$
a_4	3	$\{a_1\}$	1	$\{t_1, t_2, t_3\}$	$executandoReduce$	1000	$falhaTaskTracker$
a_5	4	$\{a_2\}$	1	$\{t_0\}$		10000	$validaResultado$
a_6	5	$\{a_1\}$	1	$\{t_1, t_2, t_3\}$		1000	$paraTaskTracker$
a_7	6	$\{a_0\}$	1	$\{t_0\}$		1000	$paraJobTracker$

tanto, as ações $\{a_2, a_3, a_4\}$ são executadas em paralelo, pois possuem o mesmo nível hierárquico, $h = 3$. A ação a_0 é executada apenas pelo testador t_0, que inicia o componente *JobTracker*. A ação a_1 é executada pelos testadores $\{t_1, t_2, t_3\}$ para iniciarem cada componente *TaskTracker*. A seguir, são executadas três ações em paralelo:

— A ação a_2 é executada pelo testador t_0, que submete a aplicação *WordCount* ao Hadoop;

— Sejam os parâmetros da ação a_3: $n_{a_3} = 1$ e $W_{a_3} = executandoMap$; a ação a_3 é executada somente pelo primeiro testador de $\{t_1, t_2, t_3\}$ que realiza uma tarefa *Map*. Deste modo, o testador injeta uma falha no componente do Hadoop por ele controlado.

— Sejam os parâmetros da ação a_4: $n_{a_4} = 1$ e $W_{a_4} = executandoReduce$; a ação a_4 é executada somente pelo primeiro testador de $\{t_1, t_2, t_3\}$ que realiza uma tarefa *Reduce*. Deste modo, o testador injeta uma falha no componente do Hadoop por ele controlado.

As funções *executandoMap* e *executandoReduce* foram implementadas utilizando os arquivos de registro (*log*) do Hadoop. São funções que ficam aguardando até a ocorrência de uma determinada sequência de caracteres, uma para cada função.

Após a execução da aplicação e as injeções de falhas, a ação a_5 é executada pelo testador t_0. Nesta ação o resultado obtido pela aplicação *WordCount* é comparado com o valor esperado, resultando em *SUCESSO* se os valores forem iguais, ou *FALHO* caso contrário. As próximas

ações finalizam a execução do Hadoop. Como dois componentes *TaskTracker* foram finalizados através das injeções de falhas ocorridas nas ações a_3 e a_4, a próxima ação, a_6, é executada por apenas um testador ($n_{a_6} = 1$) do conjunto $\{t_1, t_2, t_3\}$. A ação a_7 é executada pelo testador t_0, que finaliza o *JobTracker*.

Considerando o primeiro conjunto de máquinas para a execução do caso de falha apresentado, o testador t_1 injetou uma falha no seu *TaskTracker* porque ele foi o primeiro a executar uma tarefa *Map*. Posteriormente, o testador t_2 injetou uma falha no seu *TaskTracker* porque ele foi o primeiro a executar uma tarefa *Reduce*. O veredito deste caso de falha foi *PASSOU*, uma vez que o resultado obtido foi de acordo com o esperado, e o Hadoop se comportou de acordo com o especificado.

Considerando um outro conjunto de máquinas, o Hadoop distribuiu as tarefas de uma forma diferente da que foi realizada na execução anterior, mas o HadoopTest controlou o caso de falha de acordo com essa execução. Embora o veredito do caso de falha também tenha sido *PASSOU*, o testador t_2 foi o primeiro a injetar a falha no seu *TaskTracker* porque ele foi o primeiro a executar uma tarefa *Map*. Na execução com o conjunto anterior de máquinas o testador t_1 foi o primeiro a injetar uma falha. O Hadoop distribuiu as tarefas usando diferentes máquinas para os dois conjuntos, mas o HadoopTest controlou com sucesso cada componente do Hadoop de forma dinâmica. Além disso, o HadoopTest injetou a falha de acordo com as etapas de processamento de cada componente, e ainda validou corretamente o comportamento do Hadoop.

5.3.2 Controle Estendido

Durante a execução manual de alguns experimentos, foi identificado que o Hadoop encaminha uma tarefa *Map* de um *TaskTracker* falho sem considerar a localização do dado de entrada. Ou seja, havendo o dado de entrada em dois *TaskTrackers*, x e y, se x falhar enquanto executa uma tarefa *Map*, esta tarefa pode ser encaminhada tanto para y, que possui o dado de entrada, quanto para z, que não possui. Embora esse comportamento não seja modelado pelo mecanismo de tolerância a falhas do MapReduce, o HadoopTest foi validado

mediante a execução de dois casos de falha para esse fim.

Os casos de falha descritos nesta seção são úteis para provar a alta controlabilidade do HadoopTest, pois estendem a possibilidade e complexidade para o controle dos componentes do Hadoop. Ambos casos de falha possuem quatro testadores, um para o *JobTracker* e três para os *TaskTrackers*.

A Tabela 5.2 apresenta as ações do primeiro caso de falha, onde dois testadores injetam falhas nos seus *TaskTrackers*, mas estes precisam estar com o dado de entrada. Essa verificação é feita a partir do comando *execMapComEntrada*, atributo da ação a_3. O comando *execMapComEntrada* verifica nos arquivos de *log* onde foi replicado o dado de entrada no *DataNode* e se o *TaskTracker* está executando uma operação de Map.

Tabela 5.2: Primeiro caso de falha para a validação do controle estendido do HadoopTest.

	h	D	n	T'	W	θ	I
a_0	1	$\emptyset$	1	$\{t_0\}$		9000	$iniciaJobTracker$
a_1	2	$\{a_0\}$	3	$\{t_1, t_2, t_3\}$		1000	$iniciaTaskTracker$
a_2	3	$\{a_1\}$	1	$\{t_0\}$		900000	$submeteWordCount$
a_3	3	$\{a_1\}$	2	$\{t_1, t_2, t_3\}$	$execMapComEntrada$	1000	$falhaTaskTracker$
a_4	4	$\{a_2\}$	1	$\{t_0\}$		10000	$validaResultado$
a_5	5	$\{a_1\}$	1	$\{t_1, t_2, t_3\}$		1000	$paraTaskTracker$
a_6	6	$\{a_0\}$	1	$\{t_0\}$		1000	$paraJobTracker$

A Tabela 5.3 apresenta as ações do segundo caso de falha, onde um *TaskTracker* que falha possui o dado de entrada enquanto que o outro não, ou seja, o dado de entrada permanece disponível no outro *TaskTracker* ativo. Essa verificação é feita a partir dos comandos *execMapComEntrada* e *execMapSemEntrada*, atributos da ação a_3 e a_4, respectivamente. O comando *execMapSemEntrada* verifica nos arquivos de *log* se o *DataNode* não possui o dado de entrada e se o *TaskTracker* está executando uma operação de Map.

O HadoopTest retorna o veredito *PASSOU* no primeiro caso de falha, uma vez que o Hadoop interrompe e retorna um erro informando que o dado de entrada não se encontra mais disponível. Entretanto, no segundo caso, o veredito obtido é *FALHOU*, pois o Hadoop interrompeu a execução ainda que o dado de entrada estava disponível no *TaskTracker* ativo. O comportamento esperado era que o Hadoop encaminhasse as tarefas para o *TaskTracker*

Tabela 5.3: Segundo caso de falha para a validação do controle estendido do HadoopTest.

	h	D	n	T'	W	θ	I
a_0	1	$\emptyset$	1	$\{t_0\}$		9000	$iniciaJobTracker$
a_1	2	$\{a_0\}$	3	$\{t_1, t_2, t_3\}$		1000	$iniciaTaskTracker$
a_2	3	$\{a_1\}$	1	$\{t_0\}$		900000	$submeteWordCount$
a_3	3	$\{a_1\}$	1	$\{t_1, t_2, t_3\}$	$execMapComEntrada$	1000	$falhaTaskTracker$
a_4	3	$\{a_1\}$	1	$\{t_1, t_2, t_3\}$	$execMapSemEntrada$	1000	$falhaTaskTracker$
a_5	4	$\{a_2\}$	1	$\{t_0\}$		10000	$validaResultado$
a_6	5	$\{a_1\}$	1	$\{t_1, t_2, t_3\}$		1000	$paraTaskTracker$
a_7	6	$\{a_0\}$	1	$\{t_0\}$		1000	$paraJobTracker$

ativo, mas o Hadoop não o fez devido a uma corrupção de um arquivo de controle.

5.4 Medição Temporal

A medição temporal é uma propriedade requerida para uma plataforma de teste, que define a sua habilidade em adquirir e usar as informações temporais de um sistema durante a realização do teste. O HadoopTest possibilita testar o Hadoop considerando as suas informações temporais, seja para validar o comportamento de seus componentes ou do sistema como um todo. Como exemplo, o HadoopTest possibilita validar a latência na detecção de uma falha de um componente, ou da impossibilidade em executar uma aplicação.

A seguir, são apresentados alguns casos de falha que possibilitaram validar a medição temporal do HadoopTest. Foi identificado que o Hadoop não utiliza um atributo de configuração que define a latência na detecção de uma tarefa falha, e que ele não interrompe o processamento mesmo quando todos os seus *TaskTrackers* falham.

A Tabela 5.4 apresenta as ações do primeiro caso de falha, que possui três testadores e apenas um injeta uma falha no seu *TaskTracker* enquanto esse executa uma tarefa *Map*. O objetivo é validar se o tempo de latência especificado para que o Hadoop identifique uma tarefa falha é utilizado para reescaloná-la para um outro *TaskTracker*. O tempo de latência de uma tarefa falha é configurável através do atributo $mapreduce.task.timeout$, presente no arquivo de configuração do Hadoop, *mapred-site.xml*. Este atributo foi alterado para o tempo de um minuto, mas o Hadoop gastou cerca de treze minutos para reencaminhar a tarefa falha para

um outro *TaskTracker* ativo. Assim, o veredito desse caso de falha foi *FALHOU*, informando que o Hadoop apresentou um erro ao não reescalonar uma tarefa falha dentro da latência especificada.

Tabela 5.4: Primeiro caso de falha para a validação do controle temporal do HadoopTest.

	h	D	n	T'	W	θ	I
a_0	1	$\emptyset$	1	$\{t_0\}$		9000	$iniciaJobTracker$
a_1	2	$\{a_0\}$	3	$\{t_1, t_2, t_3\}$		1000	$iniciaTaskTracker$
a_2	3	$\{a_1\}$	1	$\{t_0\}$		900000	$submeteWordCount$
a_3	3	$\{a_1\}$	1	$\{t_1, t_2, t_3\}$	$executaMap$	1000	$falhaTaskTracker$
a_4	4	$\{a_2\}$	1	$\{t_0\}$		10000	$validaResultado$
a_5	5	$\{a_1\}$	2	$\{t_1, t_2, t_3\}$		1000	$paraTaskTracker$
a_6	6	$\{a_0\}$	1	$\{t_0\}$		1000	$paraJobTracker$

Foi alterado um outro atributo presente no arquivo de configuração do Hadoop, o atributo $mapreduce.jobtracker.expire.trackers.interval$. Este atributo define o intervalo de tempo para que o *JobTracker* verifique se os *TaskTrackers* estão ativos, ou seja, a latência para identificar um *TaskTracker* falho e reescalonar suas tarefas para outro. O atributo citado foi alterado para um minuto e o mesmo caso de falha anterior foi executado. O veredito obtido foi *PASSOU*, uma vez que após um minuto o *TaskTracker* falho foi identificado, e suas tarefas foram encaminhadas para um outro *TaskTracker* ativo.

A Tabela 5.5 apresenta as ações do segundo caso de falha, que também é formado de três testadores: um para controlar o *JobTracker* e dois para controlar os *TaskTrackers* que armazenam o dado de entrada. O objetivo é validar o comportamento do Hadoop quando o dado de entrada não está mais disponível e não existe mais qualquer *TaskTracker* ativo. Para isso, os dois testadores que controlam os *TaskTrackers* injetam uma falha nos seus *TaskTrackers* enquanto eles executam uma tarefa *Map*. Como não existe um outro *TaskTracker* ativo, o comportamento correto do Hadoop seria interromper a execução da aplicação e retornar um erro, uma vez que não existe o dado de entrada e nem um outro *TaskTracker* para executar as tarefas da aplicação. Mas isso não ocorre. O veredito desse caso de falha foi *FALHOU*, pois o Hadoop não interrompe a execução da aplicação por até treze horas, quando o HadoopTest interrompe o caso de falha dentro do tempo limite.

Tabela 5.5: Segundo caso de falha para a validação do controle temporal do HadoopTest.

	h	D	n	T'	W	θ	I
a_0	1	$\emptyset$	1	$\{t_0\}$		9000	$iniciaJobTracker$
a_1	2	$\{a_0\}$	3	$\{t_1, t_2\}$		1000	$iniciaTaskTracker$
a_2	3	$\{a_1\}$	1	$\{t_0\}$		900000	$submeteWordCount$
a_3	3	$\{a_1\}$	2	$\{t_1, t_2\}$	$executaMap$	1000	$falhaTaskTracker$
a_4	4	$\{a_2\}$	1	$\{t_0\}$		10000	$validaResultado$
a_5	5	$\{a_0\}$	1	$\{t_0\}$		1000	$paraJobTracker$

5.5 Não-Intrusividade

A propriedade de não-intrusividade de uma plataforma de teste refere-se ao nível que ela evita ou minimiza qualquer interferência não desejável ao comportamento do sistema em teste. A alteração do código fonte e o impacto no desempenho do sistema em teste são características importantes, e que ajudam a quantificar a não-intrusividade de uma plataforma em teste. O HadoopTest apresenta um alto nível de não-intrusividade uma vez que não é necessário alterar o código fonte do Hadoop para executar casos de falha, e o seu impacto sobre o desempenho é mínimo.

Como tanto o HadoopTest quanto o Hadoop são implementados em Java, a primeira versão do HadoopTest foi desenvolvida considerando a instanciação de objetos do Hadoop no próprio código do HadoopTest. Isto é, os testadores instanciam objetos *JobTracker* e *TaskTrackers* que estão disponíveis pela importação das bibliotecas do Hadoop. Além desses objetos ainda eram necessárias várias outras instanciações e parametrizações para que o Hadoop executasse. Embora ainda fosse uma abordagem não-intrusiva, o HadoopTest apresentou sobrecarga no tempo de execução ao controlar o Hadoop na execução da aplicação PiEstimator.

A Figura 5.1 mostra o tempo de execução da aplicação PiEstimator ao ser executada no Hadoop e na primeira versão do HadoopTest. O objetivo é avaliar a intrusividade causada pelo HadoopTest ao controlar o Hadoop durante a execução de uma aplicação. Os resultados e os tempos das execuções com o Hadoop são comparados com os obtidos pelo HadoopTest. O PiEstimator foi executado pelo Hadoop variando o número de instâncias da função *Map* em cada execução, e o resultado e o tempo médio das execuções foram obtidos. O HadoopTest

executou os casos de falha criados com os parâmetros das execuções anteriores, um para cada número de instância e resultado obtido. Todos os casos de falha obtiveram veredito *PASSOU*, ou seja, o HadoopTest não interferiu nos resultados obtidos, pois esses foram iguais aos obtidos pelo Hadoop. Mas, o HadoopTest apresentou aproximadamente 40% de sobrecarga no tempo de execução médio, quando comparado com o Hadoop.

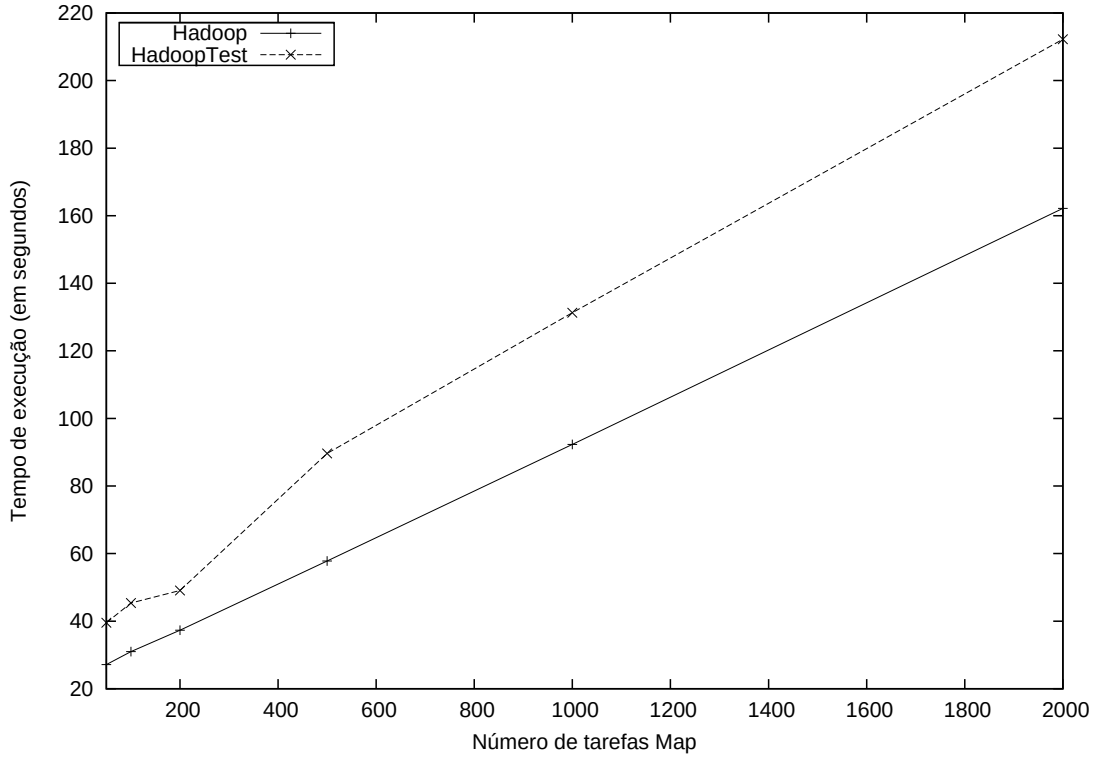


Figura 5.1: Variação do tempo de execução entre Hadoop e HadoopTest.

Embora o impacto temporal do HadoopTest pudesse ser irrelevante ao executar alguns testes, uma nova versão do HadoopTest foi desenvolvida para melhorar tanto a sobrecarga quanto a manutenibilidade. A segunda versão do HadoopTest usa os próprios *scripts* do Hadoop para inicializar e finalizar os seus componentes, usa os registros locais (*logs*) como gatilho para ativar a injeção de falhas, e usa o comando *bash kill* para injetar a falha de interrupção. Essa abordagem, ao invés da instanciação dos objetos e da instrumentação do

código fonte, melhora a sobrecarga e a manutenibilidade, o que possibilita testar outras versões do Hadoop sem se preocupar com detalhes da sua implementação. A classe *AbstractHadoop* provê as funções citadas e pode ser implementada de acordo com a necessidade. Como exemplo, a classe *AbstrackHadoop* pode usar uma outra ferramenta de injeção de falhas ou usar a programação orientada a aspecto [Kiczales et al., 1997] para ativar a injeção de falhas.

A sobrecarga do HadoopTest ao executar casos de falha foi novamente medida comparando duas execuções da aplicação PiEstimator. A Figura 5.2 mostra a média de execução do PiEstimator sendo executado diretamente pelo Hadoop e através do controle do HadoopTest. Desta vez, foram utilizadas 10, 50 e 100 máquinas no Grid'5000 para realizar as execuções.

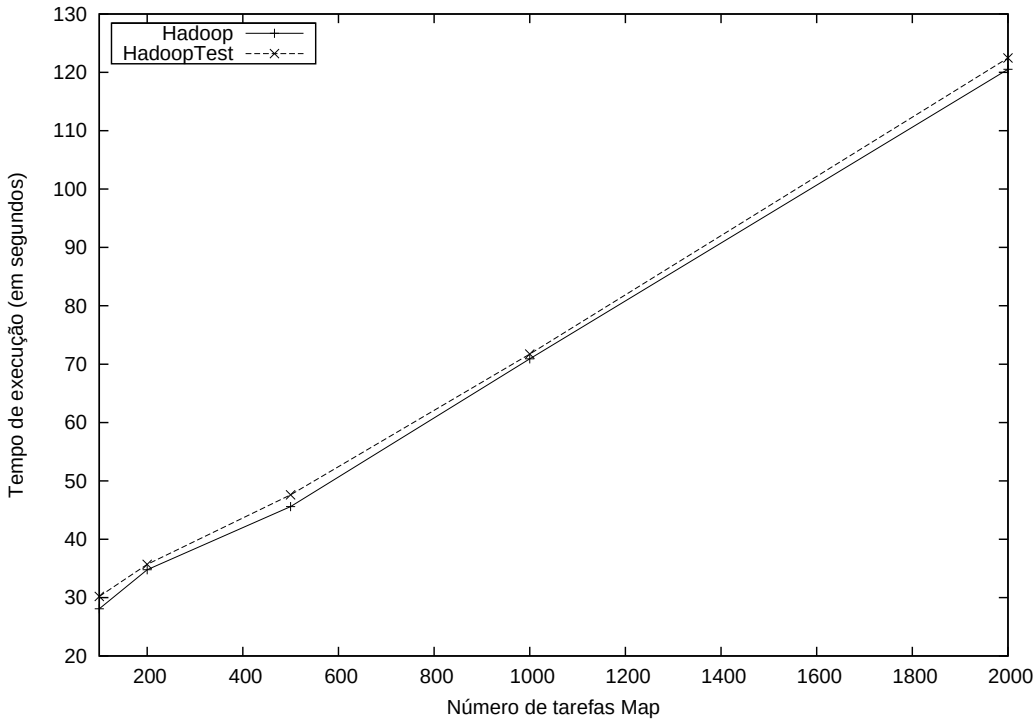


Figura 5.2: Variação do tempo de execução entre Hadoop e HadoopTest, segunda versão.

O HadoopTest confirmou a sua não-intrusividade não causando interferências nos resultados: todos os resultados obtidos via Hadoop foram iguais aos obtidos via HadoopTest, e apresentou uma interferência temporal mínima ao controlar o Hadoop durante a sua execução

dos casos de falha.

5.6 Repetibilidade e Reprodutibilidade

A repetibilidade é uma propriedade de uma plataforma de teste que define a sua habilidade em reproduzir precisamente a execução de casos de falha. Essa propriedade é altamente desejável uma vez que além de desejar encontrar um erro no SUT, é necessário repetir a execução que gerou esse erro para possibilitar a identificação e remoção do(s) defeito(s) gerador(es) do erro. A reprodutibilidade é a habilidade para reproduzir os resultados estatísticos da execução de casos de falha considerando uma determinada configuração do ambiente de execução. Normalmente, a repetibilidade implica em reprodutibilidade, uma vez que se uma execução for precisamente reproduzida, os seus resultados também serão os mesmos.

O HadoopTest apresenta uma alta repetibilidade e reprodutibilidade, uma vez que todas as execuções dos casos de falha apresentados neste trabalho são completamente repetíveis e reproduzíveis, considerando a mesma configuração do ambiente. Foram executados pelo menos três vezes cada caso de falha apresentado e os resultados obtidos foram os mesmos em todas as execuções.

5.7 Eficácia

A eficácia é a propriedade de uma plataforma de teste que diz respeito à sua habilidade em produzir um número limitado de casos de falha não-representativos. Um caso de falha não representativo ocorre quando, por exemplo, uma falha é injetada em um componente que não está sendo utilizado durante a execução do caso de falha. Esse comportamento pode ocorrer principalmente nas abordagens que usam a injeção de falhas aleatórias, ou até mesmo na abordagem desenvolvida por engenheiros de teste, como foi apresentado nos trabalhos relacionados com a avaliação da dependabilidade de sistemas MapReduce. Nos testes realizados, os casos de falha envolvem a remoção de algumas máquinas durante um período de tempo, mas não levam em consideração as etapas de processamento dos componentes que estão em execução

nas máquinas.

Como o HadoopTest não produz os casos de falha, a avaliação da propriedade de eficácia recai sobre a abordagem apresentada neste trabalho para a geração dos casos de falha. Essa abordagem possui uma alta eficácia, uma vez que são gerados apenas os casos de falha representativos para o teste de dependabilidade.

Os casos de falha gerados são ditos representativos pois são obtidos a partir do modelo do mecanismo de tolerância a falhas do MapReduce. A execução dos casos de falha representativos garante a validação da dependabilidade do sistema considerando o seu mecanismo de tolerância a falhas. Além disso, a abordagem apresentada é eficaz pois diversos erros foram identificados no Hadoop a partir da execução automática dos casos de falha representativos pelo HadoopTest.

CAPÍTULO 6

CONCLUSÃO E TRABALHOS FUTUROS

Realizar o teste de dependabilidade de um sistema MapReduce, é uma tarefa imprescindível para que a tolerância a falhas desse sistema seja garantida. Esta tese expôs e analisou os problemas relacionados com essa tarefa. Além de ser uma das primeiras propostas conhecidas para o teste de dependabilidade, este trabalho apresentou uma modelagem do mecanismo de tolerância a falhas do MapReduce, um processo para gerar casos de falha representativos a partir deste modelo, e uma plataforma de teste capaz de automatizar a execução de casos de falha em um ambiente distribuído.

Os casos de falha foram gerados a partir de um modelo formal do mecanismo de tolerância a falhas do MapReduce. Dois modelos formais foram analisados, e a modelagem em rede de Petri mostrou a melhor capacidade de representar componentes e comportamentos paralelos. Além disso, a modelagem usando redes de Petri possibilitou a apresentação de uma nova abordagem para modelar os componentes de um sistema distribuído. Os componentes do MapReduce são modelados como itens dinâmicos, possibilitando que eles sejam inseridos e removidos facilmente. Outra vantagem da nova abordagem é não precisar especificar quais estados e quais ações cada componente possui.

Um processo para a geração dos casos de falha representativos também foi apresentado. Ele é baseado na construção de um grafo de alcançabilidade da rede de Petri que modela o mecanismo de tolerância a falhas. Cada caminho nesse grafo é um caso de falha representativo que deve ser executado para a realização do teste de dependabilidade.

A plataforma de teste HadoopTest foi apresentada como uma solução para o problema da automatização da execução dos casos de falha representativos. O HadoopTest controla e monitora individualmente cada componente distribuído do Hadoop, garante que as falhas são injetadas nos componentes e nos momentos especificados, e valida o comportamento do

sistema de acordo com o esperado. Além disso, o HadoopTest utiliza uma forma simplificada para a descrição dos casos de falha e é distribuído gratuitamente para ser utilizado e adaptado para o teste de outros sistemas.

Os resultados experimentais obtidos mostraram que os casos de falha gerados a partir do modelo são representativos para o teste do sistema Hadoop, principal implementação de código aberto do MapReduce. Além disso, diversos erros foram encontrados no Hadoop, e demonstraram a alta eficácia da geração dos casos de falha, uma vez que são gerados apenas os casos representativos para o teste de dependabilidade.

Os experimentos realizados também comprovaram que o HadoopTest automatiza a execução dos casos de falha representativos, e apresenta importantes propriedades para uma plataforma de teste. O HadoopTest apresenta uma alta controlabilidade pois controla em qual componente e em qual fase do processamento as falhas são injetadas. O comportamento dos componentes e do sistema podem ser validados considerado as suas informações temporais. O sistema em teste não é alterado para executar os casos de falha, e o impacto no seu desempenho é mínimo. O HadoopTest ainda reproduz precisamente a execução de casos de falha, tarefa que envolve reproduzir os resultados estatísticos da execução. Além disso, o HadoopTest é efetivo ao realizar o teste de sistemas e identifica os sistemas que apresentam erros.

A abordagem apresentada neste trabalho para o teste de dependabilidade pode ser utilizada para testar outros sistemas MapReduce, ou ainda outros sistemas distribuídos. Os sistemas que possuem um mecanismo de tolerância a falhas equivalente ao modelado podem utilizar o mesmo conjunto obtido de casos de falha representativos. Já os sistemas que possuem um mecanismo de tolerância a falhas diferente do modelado, podem ser testados utilizando a mesma abordagem apresentada para a modelagem e a geração de casos de falha representativos. Além do mais, o HadoopTest disponibiliza uma forma facilitada para automatizar a execução de casos de falha para o teste de outros sistemas.

Trabalhos Futuros

Os trabalhos futuros podem ser classificados em três grupos de atuação: o primeiro é automatizar o processo do teste de dependabilidade; o segundo é enriquecer o modelo do mecanismo de tolerância a falhas; e o terceiro é testar outros sistemas distribuídos usando as soluções apresentadas neste trabalho.

Para automatizar o processo de teste é necessário gerar automaticamente os casos de falhas representativos a partir da rede de Petri que modela o mecanismo de tolerância a falhas do MapReduce. O objetivo é adicionar esse processo ao HadoopTest, possibilitando que um usuário forneça o modelo do mecanismo em uma linguagem de descrição para a rede de Petri – por exemplo a PNML [Pnml, 2013, Billington et al., 2003] – e o HadoopTest gere e execute os casos de falha, informando os seus vereditos.

Além de automatizar a geração dos casos de falha, também pretende-se identificar automaticamente a quantidade requerida de componentes para a completa validação de um mecanismo de tolerância a falhas. O objetivo é obter do modelo do mecanismo a quantidade de componentes necessária para que todos os comportamentos possíveis sejam testados. Por exemplo, dado um modelo qualquer, cinco componentes são necessários para que todos os casos de falha representativos sejam obtidos e executados para a completa validação do mecanismo. Esse trabalho tornará o processo de teste ainda mais efetivo, pois a quantidade de casos de falha gerada depende do número de componentes, e este pode ser definido de forma incorreta no modelo. Adicionalmente, essa automatização facilitará a modelagem que passará a requisitar apenas as diferentes ações e estados que os componentes possuem.

Alguns erros no Hadoop foram identificados a partir da execução de variações dos casos de falha representativos obtidos a partir do modelo. Assim, também é um objetivo futuro enriquecer o modelo do mecanismo de tolerância a falhas para compreender outros comportamentos que são implícitos ao mecanismo, por exemplo, o suporte às características temporais. Essa extensão deverá ser repassada para todo o processo de geração e execução dos casos de falha. Entretanto, essa tarefa ampliará a possibilidade de emprego das características temporais para

a validação de outros mecanismos e propriedades, como o teste de desempenho do sistema mediante falhas.

Por fim, a realização do teste de dependabilidade de outros sistemas distribuídos também é um desdobramento futuro. O objetivo é analisar e comprovar a possibilidade de aplicação das abordagens e soluções empregadas neste trabalho para o teste de outros sistemas, tais como o HadoopDB [Abouzeid et al., 2009] e o Hive [Thusoo et al., 2009], ou de outros mecanismos de tolerância a falhas, tal como o K-Exclusão Mútua Robusto [Rodrigues et al., 2012].

PUBLICAÇÕES REALIZADAS NO DOUTORADO

A lista a seguir registra os artigos publicados e os artigo submetidos durante o período deste doutorado.

1. J. E. Marynowski and A. R. Pimentel. A Dependability Testing Approach for MapReduce Systems Based on Representative Fault Cases. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, submetido em Nov. 2013.

2. J. E. Marynowski and A. R. Pimentel. HadoopTest: A Dependability Testing Framework for Hadoop. Technical report, Federal University of Paraná, pages 1-8, Aug. 2013.

3. J. E. Marynowski, A. R. Pimentel, T. S. Weber, and A. J. Mattos. Dependability Testing of MapReduce Systems. *In Proc. of the ICEIS - Intl. Conf. on Enterprise Information Systems*, pages 165-172. SciTePress, July 2013.

4. J. E. Marynowski. Towards Dependability Testing of MapReduce Systems. *In Proc. of the IPDPS - Intl. Parallel and Distributed Processing Symp., PhD Forum*, pages 2282-2285. IEEE, May 2013.

5. J. E. Marynowski, M. Albonico, E. C. de Almeida, and G. Sunyé. Testing MapReduce Based Systems. *In Proc. of the SBBD - Brazilian Symp. on Databases*, pages 9-16, Oct. 2011.

6. E. C. de Almeida, J. E. Marynowski, G. Sunyé, and P. Valduriez. Peer-Unit: A Framework for Testing Peer-to-Peer System. *In Proc. of the ASE - Intl. Conf. on Automated Software Engineering*, pages 169-170. ACM, Nov. 2010.

7. E. C. de Almeida, J. E. Marynowski, G. Sunyé, Y. Le Traon, and P. Valduriez. Efficient Distributed Test Architectures for Large-Scale Systems. *In Proc. of the ICTSS - Intl. Conf. on Testing Software and Systems*, pages 174-187. Springer, Nov. 2010.

REFERÊNCIAS

[Abouzeid et al., 2009] Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., and Rasin, A. (2009). HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933.

[Ambrosio, 2005] Ambrosio, A. M. (2005). *CoFI: Uma Abordagem Combinando Teste de Conformidade e Injeção de Falhas para Validação de Software em Aplicações Espaciais*. Thesis, Instituto Nacional de Pesquisas Espaciais (INPE).

[Ambrosio et al., 2005] Ambrosio, A. M., Mattiello-Francisco, F., Vijaykumar, N. L., de Carvalho, S. V., Santiago, V., and Martins, E. (2005). A Methodology for Designing Fault Injection Experiments as an Addition to Communication Systems Conformance Testing. In *Proc. of the Workshop on Dependable Software - Tools and Methods in the DSN*, pages 1–6. IEEE.

[Ambrosio et al., 2007] Ambrosio, A. M., Mattiello-Francisco, M. d. F., Santiago, V. A., Silva, W. P., and Martins, E. (2007). Designing Fault Injection Experiments Using State-Based Model to Test a Space Software. In *Proc. of the LADC - Latin-American Symp. on Dependable Computing*, pages 170–178. Springer.

[Ammann and Offutt, 2008] Ammann, P. and Offutt, J. (2008). *Introduction to Software Testing*. Cambridge University Press.

[Androutsellis-Theotokis and Spinellis, 2004] Androutsellis-Theotokis, S. and Spinellis, D. (2004). A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371.

[Apache, 2012] Apache (2012). Hadoop Distributed File System. http://hadoop.apache.org/hdfs/.

[Apache, 2013a] Apache (2013a). Apache MRUnit TM. http://mrunit.apache.org/.

[Apache, 2013b] Apache (2013b). Large-Scale Automated Test Framework - Herriot. https://issues.apache.org/jira/browse/HADOOP-6332.

[Apache, 2013c] Apache (2013c). The Apache Hadoop. http://hadoop.apache.org/.

[Apache, 2013d] Apache (2013d). The Apache Hadoop Users. http://wiki.apache.org/hadoop/PoweredBy/.

[Arlat et al., 2003] Arlat, J., Crouzet, Y., Karlsson, J., Folkesson, P., Fuchs, E., and Leber, G. (2003). Comparison of Physical and Software-Implemented Fault Injection Techniques. *IEEE Transactions on Computers*, 52(9):1115–1133.

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. E. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

[Avresky et al., 1996] Avresky, D., Arlat, J., Laprie, J.-C., and Crouzet, Y. (1996). Fault Injection for Formal Testing of Fault Tolerance. *IEEE Transactions on Reliability*, 45(3):443–455.

[Benso et al., 2007] Benso, A., Bosio, A., Carlo, S. D., and Mariani, R. (2007). A Functional Verification Based Fault Injection Environment. In *Proc. of the DFT - Intl. Symp. on Defect and Fault-Tolerance in VLSI Systems*, pages 114–122. IEEE.

[Bernardi et al., 2012] Bernardi, S., Merseguer, J., and Petriu, D. C. (2012). Dependability Modeling and Assessment in UML-Based Software Development. *The Scientific World Journal*, 2012(1):1–11.

[Bessani et al., 2010] Bessani, A. N., Cogo, V. V., Correia, M., Costa, P., Pasin, M., Silva, F., Arantes, L., Marin, O., Sens, P., and Sopena, J. (2010). Making Hadoop MapReduce Byzantine Fault-Tolerant. In *Proc. of the DSN - Intl. Conf. on Dependable Systems and Networks*, pages 1–2. IEEE.

[Billington et al., 2003] Billington, J., Christensen, S. r., van Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., and Weber, M. (2003). The Petri Net Markup Language: Concepts, Technology, and Tools. In *Proc. of the ICATPN - Intl. Conf. on Application and Theory of Petri Nets*, pages 483–505. Springer.

[Butnaru et al., 2007] Butnaru, B., Dragan, F., Gardarin, G., Manolescu, J., Nguyen, B., Pop, R., Preda, N., and Yeh, L. (2007). P2PTester: A Tool for Measuring P2P Platform Performance. In *Proc. of the ICDE - Intl. Conf. on Data Engineering*, pages 1501–1502. IEEE.

[Callou et al., 2012] Callou, G., Maciel, P., Tutsch, D., and Araújo, J. (2012). A Petri Net-Based Approach to the Quantification of Data Center Dependability. In *Petri Nets - Manufacturing and Computer Science*, chapter 14, pages 313–336. InTech.

[Chaiken et al., 2008] Chaiken, R., Jenkins, B., Larson, P.-A. k., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. (2008). Scope: Easy and Efficient Parallel Processing of Massive Data Sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276.

[Chandra et al., 2007] Chandra, T. D., Griesemer, R., and Redstone, J. (2007). Paxos Made Live: An Engineering Perspective (2006 Invited Talk). In *Proc. of the PODC - Symp. on Principles of Distributed Computing*, pages 398–407. ACM.

[Chang et al., 2008] Chang, F., Dean, J., Ghemawat, S., and Hsieh, W. (2008). Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):1–26.

[Costa et al., 2011] Costa, P., Pasin, M., Bessani, A. N., and Correia, M. (2011). Byzantine Fault-Tolerant MapReduce: Faults Are Not Just Crashes. In *Proc. of the CloudCom - Intl. Conf. on Cloud Computing Technology and Science*, pages 32–39. IEEE.

[Csallner et al., 2011] Csallner, C., Fegaras, L., and Li, C. (2011). New Ideas Track: Testing MapReduce-Style Programs. In *Proc. of the ESEC/FSE - SIGSOFT Symp. and the European Conf. on Foundations of Software Engineering*, pages 504–507. ACM.

[de Almeida, 2009] de Almeida, E. C. (2009). *Testing and Validation of Peer-to-Peer Systems*. Thesis, Université de Nantes.

[de Almeida et al., 2010a] de Almeida, E. C., Marynowski, J. E., Sunyé, G., Le Traon, Y., and Valduriez, P. (2010a). Efficient Distributed Test Architectures for Large-Scale Systems. In *Proc. of the ICTSS - Intl. Conf. on Testing Software and Systems*, pages 174–187. Springer.

[de Almeida et al., 2010b] de Almeida, E. C., Marynowski, J. E., Sunyé, G., and Valduriez, P. (2010b). PeerUnit: A Framework For Testing Peer-to-Peer Systems. In *Proc. of the ASE - Intl. Conf. on Automated Software Engineering*, pages 169–170. ACM.

[de Almeida et al., 2008] de Almeida, E. C., Sunyé, G., Traon, Y. L., and Valduriez, P. (2008). A Framework for Testing Peer-to-Peer Systems. In *Proc. of the ISSRE - Intl. Symp. on Software Reliability Engineering*, pages 167–176. IEEE.

[de Almeida et al., 2010c] de Almeida, E. C., Sunyé, G., Traon, Y. L., and Valduriez, P. (2010c). Testing Peer-to-Peer Systems. *ESE - Empirical Software Engineering*, 15(4):346–379.

[Dean and Ghemawat, 2004] Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the OSDI - Symp. on Operating Systems Design and Implementation*, pages 137–149. USENIX.

[DeWitt et al., 2008] DeWitt, D. J., Paulson, E., Robinson, E., Naughton, J., Royalty, J., Shankar, S., and Krioukov, A. (2008). Clustera: An Integrated Computation And Data Management System. *Proceedings of the VLDB Endowment*, 1(1):28–41.

[Echtle and Leu, 1994] Echtle, K. and Leu, M. (1994). Test of Fault Tolerant Distributed Systems by Fault Injection. In *Proc. of the FTPDS - Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 244–251. IEEE.

[El-Fakih et al., 2009] El-Fakih, K., Yevtushenko, N., and Fouchal, H. (2009). Testing Timed Finite State Machines with Guaranteed Fault Coverage. In *Proc. of the TESTCOM/FATES - Testing of Software and Communication Systems*, pages 66–80. Springer.

[Eventbrite, 2013] Eventbrite (2013). Hadoop in Finance Day Chicago. http://hadoopinfinancechicago.eventbrite.com/.

[Forbes, 2013] Forbes (2013). Morgan Stanley Takes On Big Data With Hadoop. http://www.forbes.com/sites/tomgroenfeldt/2012/05/30/morgan-stanley-takes-on-big-data-with-hadoop/.

[Fu et al., 2004] Fu, C., Ryder, B. G., Milanova, A., and Wonnacott, D. (2004). Testing of JavaWeb Services for Robustness. In *Proc. of the ISSTA - Intl. Symp. on Software Testing and Analysis*, pages 23–33. ACM.

[Ghemawat et al., 2003] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The Google File System. In *Proc. of the SOPS - Symp. on Operating Systems Principles*, pages 29–43. ACM.

[Grid5000, 2013] Grid5000 (2013). Grid'5000. http://www.grid5000.fr/.

[Group, 2013] Group, O. M. (2013). Unified Modeling Language (UML). http://www.omg.org/spec/UML/.

[Gu and Grossman, 2009] Gu, Y. and Grossman, R. L. (2009). Sector and Sphere: The Design and Implementation of a High Performance Data Cloud. *Philosophical Transactions A - Royal Society*, 367(1897):2429–2445.

[Gunawi, 2009] Gunawi, H. S. (2009). *Towards Reliable Storage Systems*. Thesis, University of Wisconsin-Madison.

[Gunawi et al., 2011] Gunawi, H. S., Do, T., Joshi, P., Alvaro, P., Hellerstein, J. M., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Sen, K., and Borthakur, D. (2011). FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proc. of the NSDI - Conf. on Networked Systems Design and Implementation*, pages 1–14. USENIX.

[Gunawi et al., 2010] Gunawi, H. S., Do, T., Joshi, P., Hellerstein, J. M., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Sen, K. (2010). Towards Automatically Checking Thousands of Failures with Micro-Specification. In *Proc. of the HotDep - Intl. Conf. on Hot Topics in System Dependability*, pages 1–6. USENIX.

[Henry, 2009] Henry, A. (2009). Cloud Storage FUD: Failure, Uncertainty and Durability (Keynote Address). In *Proc. of the FAST - Symp. on File and Storage Technologies*, pages 1–33. USENIX.

[Hoarau et al., 2007] Hoarau, W., Tixeuil, S., and Vauchelles, F. (2007). FAIL-FCI: Versatile Fault Injection. *Future Generation Computer Systems*, 23(7):913–919.

[Hsueh et al., 1997] Hsueh, M.-C., Tsai, T., and Iyer, R. (1997). Fault Injection Techniques and Tools. *Computer*, 30(4):75–82.

[Huang et al., 2010] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. (2010). The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In *Proc. of the ICDEW - Intl. Conf. on Data Engineering Workshops*, pages 41–51. IEEE.

[ISO/IEC, 1994] ISO/IEC (1994). *ISO/IEC 9646-1 - Conformance Testing Methodology and Framework - Part 1: General Concepts*. International Organization for Standardization/International ElectroTechnial Commission.

[Jacques-Silva et al., 2006] Jacques-Silva, G., Drebes, R., Gerchman, J., F. Trindade, J., Weber, T., and Jansch-Porto, I. (2006). A Network-Level Distributed Fault Injector for Experimental Validation of Dependable Distributed Systems. In *Proc. of the COMPSAC - Intl. Computer Software and Applications Conference*, pages 421–428. IEEE.

[Joshi, 2012] Joshi, P. (2012). *Predictive and Programmable Testing of Concurrent and Cloud Systems*. Thesis, University of California at Berkeley.

[Joshi et al., 2011] Joshi, P., Gunawi, H. S., and Kou (2011). PREFAIL: A Programmable Tool for Multiple-Failure Injection. In *Proc. of the OOPSLA - Conf. on Object-Oriented Programming*, pages 171–188. ACM.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-m., and Irwin, J. (1997). Aspect-Oriented Programming. In *Proc. of the ECOOP - European Conf. on Object-Oriented Programming*, pages 1–25. Springer.

[Lefever et al., 2004] Lefever, R., Joshi, K., Cukier, M., and Sanders, W. (2004). A Global-State-Triggered Fault Injector for Distributed System Evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):593–605.

[Lin et al., 2010] Lin, H., Ma, X., Archuleta, J., Feng, W.-c., Gardner, M., and Zhang, Z. (2010). MOON: MapReduce On Opportunistic eNvironments. In *Proc. of the HPDC - Intl. Symp. on High Performance Distributed Computing*, pages 95–106. ACM.

[Liu et al., 2004] Liu, Y., Liu, X., Xiao, L., Ni, L. M., and Zhang, X. (2004). Location-Aware Topology Matching in P2P Systems. In *Proc. of the INFOCOM - Conf. of the Computer and Communications Societies*, pages 2220–2230. IEEE.

[Marinescu et al., 2010] Marinescu, P. D., Banabic, R., and Candea, G. (2010). An Extensible Technique for High-precision Testing of Recovery Code. In *Proc. of the USENIXATC - Annual Technical Conf.*, pages 22–23. USENIX.

[Marynowski, 2013] Marynowski, J. E. (2013). Towards Dependability Testing of MapReduce Systems. In *Proc. of the IPDPS - Intl. Parallel and Distributed Processing Symp., PhD Forum*, pages 2282–2285. IEEE.

[Marynowski et al., 2011] Marynowski, J. E., Albonico, M., de Almeida, E. C., and Sunyé, G. (2011). Testing MapReduce-Based Systems. In *Proc. of the SBBD - Brazilian Symp. on Databases*, pages 9–16.

[Marynowski and Pimentel, 2013] Marynowski, J. E. and Pimentel, A. R. (2013). HadoopTest: A Dependability Testing Framework for Hadoop. Technical report, Federal University of Paraná.

[Marynowski et al., 2013] Marynowski, J. E., Pimentel, A. R., Weber, T. S., and Mattos, A. J. (2013). Dependability Testing of MapReduce Systems. In *Proc. of the ICEIS - Intl. Conf. on Enterprise Information Systems*, pages 165–172. SciTePress.

[Murata, 1989] Murata, T. (1989). Petri nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580.

[Natella et al., 2012] Natella, R., Cotroneo, D., Duraes, J. A., and Madeira, H. S. (2012). On Fault Representativeness of Software Fault Injection. *IEEE Transactions on Software Engineering*, 39(1):80–96.

[Pan et al., 2009] Pan, X., Tan, J., Kalvulya, S., Gandhi, R., and Narasimhan, P. (2009). Blind Men and the Elephant: Piecing Together Hadoop for Diagnosis. In *Proc. of the ISSRE - Intl. Symp. on Software Reliability Engineering, Industrial Track*, pages 1–10. IEEE.

[Pan et al., 2010] Pan, X., Tan, J., Kavulya, S., Gandhi, R., and Narasimhan, P. (2010). Ganesha: Black-Box Diagnosis of MapReduce Systems. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):8–13.

[Papadimitriou and Sun, 2008] Papadimitriou, S. and Sun, J. (2008). DisCo: Distributed Co-clustering with Map-Reduce: A Case Study towards Petabyte-Scale End-to-End Mining. In *Proc. of the ICDM - Intl. Conf. on Data Mining*, pages 512–521. IEEE.

[Peres, 2010] Peres, L. M. (2010). *Proposta de um Método de Verificação por Tempo Global com Redes de Petri no Desenvolvimento de Software Embarcado e em Tempo Real*. Thesis, Federal University of Paraná.

[Pham et al., 2011] Pham, C., Chen, D., Kalbarczyk, Z., and Iyer, R. K. (2011). CloudVal: A Framework for Validation of Virtualization Environment in Cloud Infrastructure. In *Proc. of the DSN - Intl. Conf. on Dependable Systems and Networks*, pages 189–196. IEEE.

[Pisoni, 2012] Pisoni, A. (2012). Skynet - A Ruby MapReduce Framework. http://skynet.rubyforge.org/.

[Pnml, 2013] Pnml (2013). Petri Net Markup Language (PNML). http://www.pnml.org/.

[Ranger et al., 2007] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C. (2007). Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proc. of the HPCA - Intl. Symp. on High Performance Computer Architecture*, pages 13–24. IEEE.

[Rodrigues et al., 2012] Rodrigues, L. A., Duarte, E. P., and Arantes, L. (2012). Exclusão Mútua Distribuída e Robusta para k Recursos Compartilhados. In *Proc. of the WTF - Workshop de Testes e Tolerância à Falhas*, pages 43–56.

[Sangroya et al., 2012a] Sangroya, A., Serrano, D., and Bouchenak, S. (2012a). Benchmarking Dependability of MapReduce Systems. In *Proc. of the SRDS - Symp. on Reliable Distributed Systems*, pages 21–30. IEEE.

[Sangroya et al., 2012b] Sangroya, A., Serrano, D., and Bouchenak, S. (2012b). MRBS: Towards Dependability Benchmarking for Hadoop MapReduce. In *Proc. of the Euro-Par BDMC - Workshop on Big Data Management in Clouds*, pages 3–12. Springer.

[Schroeder and Gibson, 2006] Schroeder, B. and Gibson, G. A. (2006). A Large-Scale Study of Failures in High-Performance Computing Systems. In *Proc. of the DSN - Intl. Conf. on Dependable Systems and Networks*, pages 249–258. IEEE.

[Stott et al., 2000] Stott, D. T., Floering, B., Burke, D., Kalbarczpk, Z., and Iyer, R. K. (2000). NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *Proc. of the IPDPS - Intl. Parallel and Distributed Processing Symp.*, pages 91–100. IEEE.

[Tan et al., 2008] Tan, J., Pan, X., Kavulya, S., Gandhi, R., and Narasimhan, P. (2008). SALSA: Analyzing Logs as StAte Machines. In *Proc. of the WASL - Conf. on Analysis of System Logs*, pages 1–8. USENIX.

[Tan et al., 2009] Tan, J., Pan, X., Kavulya, S., Gandhi, R., and Narasimhan, P. (2009). Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop. In *Proc. of the HotCloud - Conf. on Hot Topics in Cloud Computing*, pages 1–5. USENIX.

[Team, 2012] Team, H. (2012). Fault Injection Framework and Development Guide. http://hadoop.apache.org/hdfs/docs/r0.21.0/faultinject_framework.html.

[Thusoo et al., 2009] Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009). Hive - A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629.

[Ural, 1992] Ural, H. (1992). Formal Methods for Test Sequence Generation. *Computer Communications*, 15(5):311–325.

[Voas, 1997] Voas, J. (1997). Software Fault Injection: Growing Safer Systems. In *Proc. of the AERO - Aerospace Conf.*, pages 551–561. IEEE.

[Walter et al., 1998] Walter, T., Schieferdecker, I., and Grabowski, J. (1998). Test Architectures for Distributed Systems - State of the Art and Beyond. In *Proc. of the IWTCS - Intl. Workshop on Testing of Communicating Systems*, pages 149–174. Kluwer Academic Pub.

[Wang et al., 2013] Wang, L., Tao, J., Ranjan, R., Marten, H., Streit, A., Chen, J., and Chen, D. (2013). G-Hadoop: MapReduce Across Distributed Data Centers for Data-Intensive Computing. *Future Generation Computer Systems*, 29(3):739–750.

[White, 2012] White, T. (2012). *Hadoop: The Definitive Guide*. O'Reilly Media, 3rd edition.

[Winter and Kostamaa, 2010] Winter, R. and Kostamaa, P. (2010). Large Scale Data Warehousing: Trends and Observations (Keynote Address). In *Proc. of the ICDE - Intl. Conf. on Data Engineering*, pages 1–35. IEEE.

[Yang et al., 2007] Yang, H.-c., Dasdan, A., Hsiao, R.-L., and Parker, D. (2007). Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *Proc. of the SIGMOD - Intl. Conf. on Management of Data*, pages 1029–1040. ACM.

[Zaharia et al., 2008] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. (2008). Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of the OSDI - Symp. on Operating Systems Design and Implementation*, pages 29–42. USENIX.

[Zhou et al., 2006] Zhou, Z., Wang, H., Zhou, J., Tang, L., and Li., K. (2006). Pigeon: A Framework for Testing Peer-to-Peer Massively Multiplayer Online Games over Heterogeneous Network. In *Proc. of the CCNC - Consumer Communications and Networking Conf.*, pages 1028–1032. IEEE.