MICHEL ALBONICO

HADOOPTEST: UM CONTROLADOR DE TESTES PARA SISTEMAS BASEADOS EM MAPREDUCE

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Eduardo Cunha de Almeida

CURITIBA

2011

A339

Albonico, Michel
Hadooptest : um controlador de testes para sistemas baseados em MapReduce / Michel Albonico. –
Curitiba: 2012.
65 f.
Orientação Dr. Eduardo Cunha de Almeida
Dissertação (Mestrado) - Setor de Ciências Exatas da Universidade Federal do Paraná, 2010.

Inclui bibliografia.

1.Software - testes. 2. Processamento Paralelo. I. Título.

CDD 005.2750285514

Catalogação na fonte pela Biblioteca de Ciência e Tecnologia da UFPR

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

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Michel Albônico, avaliamos o trabalho intitulado, *"HadoopTest: um Controlador de Testes Distribuídos para Sistemas baseados em MapReduce"*, cuja defesa foi realizada no dia 08 de dezembro de 2011, às 10:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 08 de dezembro de 2011.

Prof. Dr. Eduardo Cunha de Almeida
DINF/UFPR – Orientador

Prof. Dr. Gerson Sunyé
INRIA/França – Membro Externo

Prof. Dr. Marcos Didonet Del Fabro
DINF/UFPR – Membro Interno

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

Dedico este trabalho especialmente aos meus pais Claudete e Silvio que tiveram sabedoria ao me guiar e por nunca terem me deixado desistir. Também dedico à minha noiva Daiane que sempre me apoiou e pela paciência em meus momentos de indisponibilidade. Estendo minha dedicatória ao professor Eduardo Cunha de Almeida, pela sua paciência, competência, sabedoria e dedicação

AGRADECIMENTOS

Agradeço inicialmente aos companheiros João Eugênio e Edson Ramiro pela grande ajuda e participação no desenvolvimento de meu trabalho.

A Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelo bolsa de estudos.

Ao professor Gerson Sunye, pela paciência e dedicação nos momentos que precisei.

A professora e Irmã Lucila Mai, diretora da Faculdade da Fronteira, pela compreensão durante as atividades de mestrado.

E finalmente, a Universidade Federal do Paraná (UFPR), Departamento de Informática, demais professores e funcionários.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

API	*Application Programming Interface*
CTMF	*Conformance Testing Methodology and Framework*
DTS	*Declarative Testing Specification*
FTS	*Fail Testing Service*
GFS	*Google Filesystem*
HDFS	*Hadoop Distributed Filesystem*
JPDA	*Java Plataform Debugger Architecture*
JVM	*Java Virtual Machine*
MR	MapReduce
MTC	*Main Test Controller*
P2P	Peer-to-Peer
PB	Petabytes
PTC	*Partial Test Case*
RMI	*Remote Method Invocation*
SAD	Sistema de Arquivos Distribuído
SGBD	Sistema Gerenciador de Banco de Dados
SO	Sistema Operacional
SUT	*System Under Test*
TAM	*Test Application Management*
TC	*Test Controller*
TMT	*Test and Monitoring Tool*
TTCN-3	*Testing and Test Control Notation* versão 3

RESUMO

O MapReduce (MR) é uma das soluções mais populares na área de processamento de dados em grande escala. Os sistemas baseados em MapReduce frequentemente são implantados sobre agrupamentos de computadores, onde falhas acontecem constantemente, devido a defeitos de software, problemas de hardware e interrupções. Testar sistemas baseados em MapReduce é difícil, uma vez que é necessário um grande esforço do controlador de testes para executar casos de teste distribuídos em ambientes com a presença de falhas. Neste trabalho, apresentamos uma nova solução de testes para resolver isso, que foi chamada de HadoopTest. Esta solução baseia-se em uma abordagem de controle escalável, onde um coordenador gerencia diversos testadores distribuídos, que controlam os componentes do MR. Os testadores podem simular falhas sobre os componentes do MR e monitorar suas execuções. O HadoopTest foi utilizado para testar duas aplicações distribuídas juntamente com o Hadoop (i.e., a implementação MapReduce de código aberto mantida pela fundação Apache). Nossos experimentos apresentaram resultados promissores, sendo que o HadoopTest conseguiu coordenar casos de teste distribuídos, injetar falhas nos componentes do MR e encontrar alguns defeitos de software que foram propositalmente inseridos.

Palavras-chave: Teste de MapReduce; MapReduce; Processamento paralelo; HadoopTest.

ABSTRACT

MapReduce (MR) is one of the most popular solution on large-scale data processing area. The MR-based systems are often deployed over clusters of computers, where failures happen constantly due to bugs, hardware problems, and outages. Testing MR-based systems is hard, since it is needed a great effort of test controller to execute distributed test cases upon failures. In this work, we present a novel testing solution to tackle this issue called HadoopTest. This solution is based on a scalable control approach, where a coordinator manages many distributed testers which control the MR components. Testers are allowed to simulate failures on MR components and monitor their behavior. HadoopTest was used to test two applications bundled into Hadoop (i.e., a open source MapReduce implementation mantained by Apache Foundation). On our experiments HadoopTest was able to coordinate distributed test cases, inject faults on MR components and find some bugs which were purposely inserted.

Keywords: MapReduce Test; MapReduce; Parallel processing; HadoopTest.

CAPÍTULO 1

INTRODUÇÃO

Na última década, algumas aplicações, como relatórios customizados (e.g., em aplicações de Data Warehouse ou Data Mining), dados de pesquisas e redes sociais, geraram quantidades de dados na ordem de petabytes (PB) e essa quantidade tende a aumentar ao longo dos próximos anos [38]. O processamento dessas grandes quantidades de dados requer um esforço computacional elevado e tanto a comunidade acadêmica como a indústria vêm se dedicando na criação de novas tecnologias para melhorar a utilização de recursos computacionais no processamento de dados. O *framework* MapReduce (MR) [14] é uma das alternativas mais populares nesta área. Ele permite que usuários pouco experientes em computação distribuída utilizem um grande número de computadores para processamento de dados. Dentre as diversas implementações de MapReduce, o Hadoop [33], uma implementação MapReduce de código aberto mantido pela fundação Apache, é utilizado por um grande número de empresas, como: Ebay, Linkdln, Quantcast, Facebook e Yahoo [34]. Algumas empresas de banco de dados, incluindo Aster, Greenplum e Vertica, também começaram a integrar funcionalidades de grande escala em seus Sistemas Gerenciadores de Banco de Dados (SGBD) através do MapReduce.

Como outros ambientes de grande escala, as implementações do MapReduce estão sujeitas a falhas de diferentes tipos (e.g, interrupções, problemas de hardware, defeitos de software) [23]. Sendo assim, estas implementações precisam ser projetadas com tolerância a falhas e testadas, bem como devem ser testadas as aplicações criadas para serem executadas sobre o MR. No entanto, testar sistemas de grande escala é uma tarefa complexa e requer uma arquitetura de controle (i.e., *harness* de teste), que coordene a execução de casos de teste distribuídos sobre ambientes de grande escala. Uma arquitetura de controle para testar sistemas baseados em MapReduce deve, além de coordenar a execução de casos de teste distribuídos, monitorar os nós que executam o caso de teste e prover

meios para injetar falhas no sistema testado, ou seja, reproduzir falhas que possam estar presentes em um ambiente real (e.g, problemas de hardware, interrupções). Nenhuma das soluções para teste de sistemas baseados em MapReduce abordadas neste trabalho atende a estas três características de controle.

Este trabalho apresenta o HadoopTest, um controlador de testes para sistemas baseados em MapReduce. O HadoopTest gerencia a execução de casos de teste sobre os componentes (i.e., nós) de uma implementação MR. A arquitetura do HadoopTest é dividida em um coordenador e vários testadores, sendo que cada testador, inicializa, controla e monitora um dos componentes do MapReduce e executa as ações de teste sobre ele. O HadoopTest foi utilizado para testar duas aplicações distribuídas juntamente com o Hadoop (i.e., a implementação MapReduce de código aberto mantida pela fundação Apache). Nossos experimentos apresentaram resultados promissores, sendo que o HadoopTest conseguiu coordenar casos de teste distribuídos, injetar falhas nos componentes do MR e encontrar alguns defeitos de software que foram propositalmente inseridos.

O restante do trabalho é organizado da seguinte maneira. O próximo capítulo introduz conceitos básicos de MapReduce. O Capítulo 3 conceitua teste de software e compara algumas ferramentas para teste de sistemas distribuídos e para teste de sistemas baseados em MapReduce. O Capítulo 4 apresenta o HadoopTest e suas características de implementação. O Capítulo 5 mostra os experimentos e os resultados conseguidos com o HadoopTest. O Capítulo 6 conclui sobre os resultados deste trabalho.

CAPÍTULO 2

MAPREDUCE

O MapReduce é uma arquitetura para processamento de grandes quantidades de dados proposta por Dean e Ghemawat [15], que distribui o processamento sobre vários computadores. No MapReduce, a coordenação do processamento e tratamento de falhas do ambiente é feita sem a intervenção do usuário, cabendo a ele somente criar sistemas para serem executados no MapReduce.

A programação de sistemas para MapReduce baseia-se em duas funções: *map* e *reduce*. O map faz uma análise inicial dos dados de entrada, definida pelo usuário, e a função de reduce agrupa os resultados da função de map. O fluxo de execução entre as funções de map e reduce é mostrado na Figura 2.1.

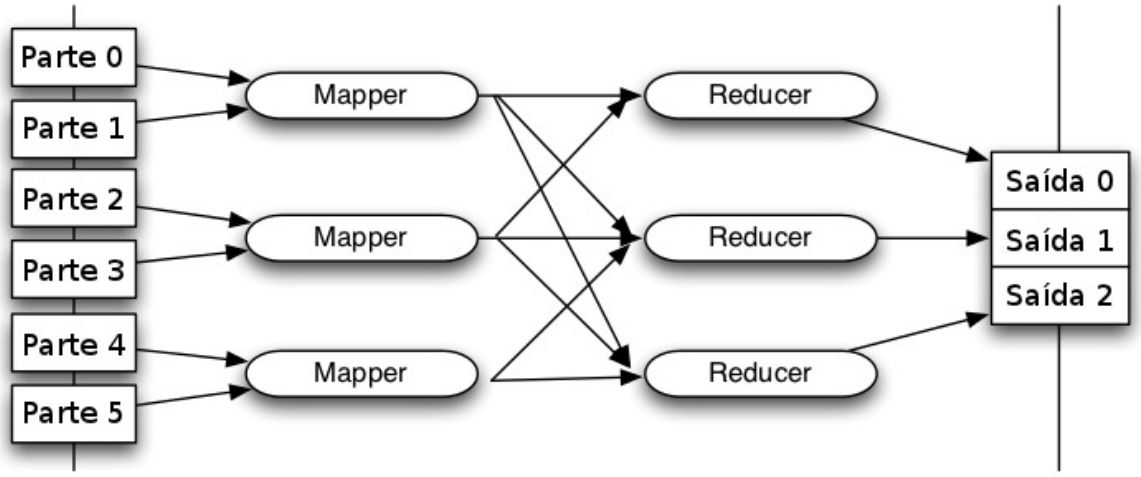


Figura 2.1: Fluxo de execução do MapReduce.

O MapReduce recebe uma entrada (e.g., um arquivo de texto) e a divide em partes menores, de tamanho pré-definido. Cada um dos *mappers* (componentes responsáveis por executar a função de map) recebe uma ou mais partes do arquivo de entrada e faz o processamento definido na função de map do sistema. Os mappers geram como saída um par composto por uma chave e um valor. Para cada chave distinta é designado um *Reducer* (componente responsável por executar a função de reduce), que faz a agregação

dos valores correspondentes à chave pela qual ele é responsável e gera uma saída. Quando todos os Reducers terminam de executar, então a saída deles é agrupada em um único arquivo, correspondendo à saída do sistema MapReduce.

2.1 Exemplo de uma Aplicação MapReduce

O WordCount é a aplicação para MapReduce utilizada como exemplo por [15]. Ele recebe um texto como entrada e conta as ocorrências de cada palavra neste texto. O pseudo-código do WordCount é mostrado na Figura 2.2.

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");
        reduce(String key, Iterator values):

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
    result += ParseInt(v);
    Emit(AsString(result));
```

Figura 2.2: Pseudo-código da aplicação WordCount.
Fonte: Dean e Ghemawat [15].

Assim como outras aplicações para MapReduce, o WordCount se divide em duas funções principais, map e reduce. Nele, a função map lê as linhas de um texto de entrada (i.e., parâmetro value) e gera como saída um par formado por uma chave (i.e., variável *w*, que corresponde a uma palavra) e um valor (i.e., um número inteiro de valor 1) e a funçao reduce recebe como parâmetro as saídas da função map, agrupa todas as palavras iguais e soma seus valores, gerando o resultado final da aplicação WordCount.

A Figura 2.3 mostra um exemplo de texto de entrada para uma aplicação WordCount.

```
Oi
UFPR
Michel Albonico
UFPR
```

Figura 2.3: Exemplo de texto de entrada para a aplicação WordCount.

O resultado de uma execução da aplicação WordCount sobre o texto mostrado na Figura 2.3 emitiria os pares mostrados na Figura 2.4.

```
< Albonico, 1 >
< Michel, 1 >
< Oi, 1 >
< UFPR, 2 >
```

Figura 2.4: Exemplo da saída da aplicação WordCount.

O resultado da execução do WordCount é uma listagem ordenada alfabeticamente pelas palavras encontradas no texto. Para cada palavra é criado um par contendo a palavra e a quantidade de vezes que ela foi encontrada.

2.2 Arquitetura de Execução do MapReduce

O MapReduce possui uma arquitetura de execução composta por um componente principal, chamado de mestre, e n outros componentes secundários, chamados de escravos. O componente mestre é quem coordena os componentes escravos e sincroniza a execução das funções de Map e Reduce dos sistemas para MapReduce. A arquitetura do MapReduce é ilustrada na Figura 2.5.

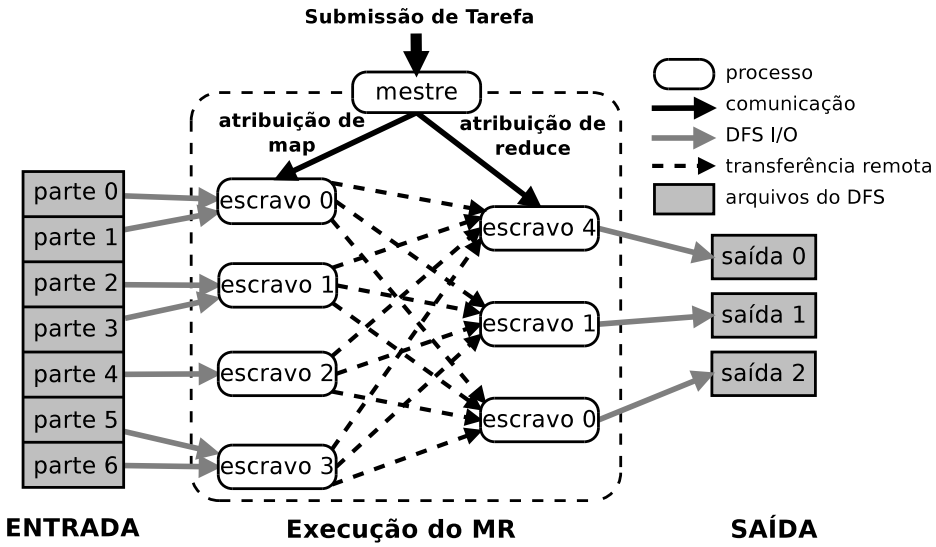


Figura 2.5: Arquitetura de execução do MapReduce.
Fonte: Figura adaptada de Dean e Ghemawat [15].

Na figura, uma tarefa (i.e., sistema para MapReduce) é submetida ao nodo mestre, juntamente com seus dados de entrada. Então o componente mestre divide os dados de entrada em partes e atribui para alguns componentes escravos a execução da função de map. Posteriormente outros componentes são designados para executarem a função de reduce. Conforme os componentes responsáveis pela função de map terminam de executar, eles respondem ao nodo mestre. Então, o nodo mestre envia uma mensagem para os componentes pela função de reduce, para que eles busquem os dados em cada um dos Mappers que já terminaram de executar. Ao término de todos os Reducers é feito um agrupamento dos seus resultados e a tarefa é finalizada.

A localidade dos dados do MapReduce acontece de duas formas, localmente e no Sistema de Arquivos Distribuído (SAD). Os dados de entrada e os resultados da função de Reduce são armazenados no Sistema de Arquivos da Google (*Google File System*, GFS) [18], que é o SAD utilizado pelo MapReduce. O GFS permite a replicação de dados e por causa dela o MapReduce consegue ter uma maior disponibilidade dos seus dados. A técnica de replicação consiste em fazer uma cópia (i.e., réplica) dos dados de um componente para outros, dessa forma, se determinado componente falhar, os dados que estavam armazenados nele permanecem disponíveis através de suas réplicas.

2.2.1 Implementações do MapReduce

A implementação original do MapReduce não é disponibilizada com o código fonte aberto, pois ela é uma implementação proprietária da Google, empresa detentora de sua patente. Existem algumas implementações MapReduce de código fonte aberto baseadas na implementação da Google, como o Hadoop, o Aster MapReduce e o Disco. O Hadoop é uma implementação mantida pela fundação Apache [1], que é utilizado por um grande número de companias, como: Ebay, Linkdln, Quantcast, Facebook e Yahoo [34]. A empresa Cloudera [2] contribui com melhorias no Hadoop. Ela possui um processo de desenvolvimento paralelo ao do Hadoop. A principal contribuição da Cloudera são as correções de defeitos no Hadoop, garantindo a sua confiabilidade, para que ele seja distribuído comercialmente [10]. O Aster MapReduce é uma implementação da empresa Aster Data [3], que utiliza o MapReduce para realizar consultas analíticas em bancos de dados relacionais e não-relacionais [11]. O Disco é uma implementação MapReduce desenvolvida pelo centro do pesquisas da Nokia [4], escrita na linguagem de programação Erlang [5] (i.e., linguagem de programação utilizada para criação de programas de tempo real, como sistemas de telefonia, de bancos e de comércio eletrônico, altamente escaláveis e disponíveis) [17].

2.2.1.1 Hadoop

O Hadoop é uma implementação MapReduce de código aberto e é utilizado por uma grande quantidade de empresas [34]. Ele é dividido em duas partes [4]: Sistema de Arquivos Distribuído do Hadoop (*Hadoop Distributed File System*, HDFS) e uma arquitetura de gerência e execução do processamento.

O HDFS é baseado no Sistema de Arquivos da Google (*Google Filesystem*, GFS), proposto em Ghemawat et al. [18]. Ele se dividide em NameNode e DataNode, sendo que o DataNode é responsável pelo armazenamento dos dados e o NameNode pelo gerenciamento dos dados e dos DataNodes.

[1] http://www.apache.org
[2] http://www.cloudera.com
[3] http://www.asterdata.com
[4] http://research.nokia.com
[5] http://www.erlang.org/

A arquitetura de execução do Hadoop é composta por um JobTracker e vários TaskTracker. O JobTracker corresponde ao componente principal do MapReduce e é ele quem gerencia os TaskTrackers e sincroniza a execução das tarefas do MapReduce. Já os TaskTrackers são responsáveis pela execução das funções de Map e de Reduce.

CAPÍTULO 3

TESTE DE SOFTWARE

Um software pode ser responsável por diversas atividades ditas como vitais à sociedade moderna. Ele é utilizado em aplicações distintas, como aeronaves, sistemas para controle de tráfego aéreo, relógios, carros, celulares e controles remotos, por exemplo. A maioria das pessoas que utilizam um software acredita que ele nunca irá falhar [3], no entanto, os softwares podem falhar devido a diversos fatores (i.e., falhas de hardware, erros de programação, entrada de dados errada). Para diminuir as chances dos softwares falharem, e com isso garantir a sua qualidade, existem algumas atividades que podem ser realizadas, dentre elas o teste de software [29].

O teste de software possui um cenário que basicamente divide-se em [16]: testador, programa, entrada, saída e resultado, conforme a Figura 3.1.

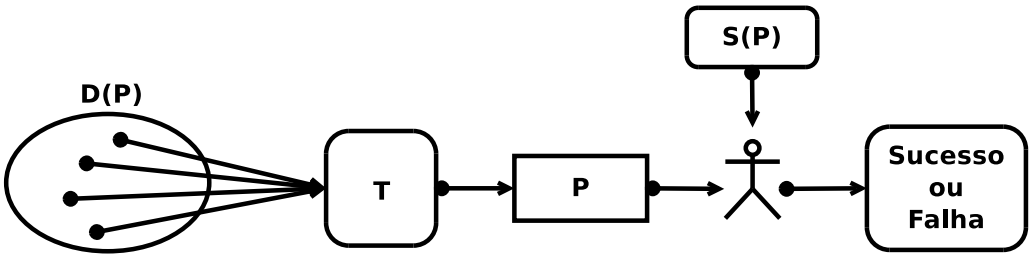


Figura 3.1: Ambiente de teste básico.
Fonte: Figura adaptada de Delamaro et al [16].

No ambiente de teste, apresentado na Figura 3.1, a entrada é descrita como Domínio de Entrada *D(P)*. Ela caracteriza-se pelo conjunto de valores válidos para a execução do programa *P*. Considerando o domínio de entrada, escolhe-se um dos casos de testes *T* a ser executado para *P*. O caso de teste é onde são definidas todas as ações de teste a serem executadas sobre um programa *P*. Após a execução e consequentemente, saída do caso de teste, são analisadas as especificações do programa *S(P)* (i.e., resultados esperados) concluindo se o teste obteve falha ou sucesso.

Para um melhor entendimento de um cenário de teste de software, algumas nomenclaturas são descritas na Seção 3.1 e na Seção 3.2 são mostrados os níveis de testes de

software.

3.1 Terminologias de Teste de Software

Nesta seção são descritas as nomenclaturas ligadas a teste de software destacadas por [3, 16, 29].

O *caso de teste* é a primeira delas. É nele que são definidas todas as ações de teste a serem executadas sobre um software. Para a execução de um caso de teste são necessários alguns valores de entrada, ditos como *valores do caso de teste*. Dependendo do conjunto de valores de entrada é esperado um resultado pré-definido (*resultado esperado*), após a execução do caso de teste. Tendo um caso de teste executado, o resultado de sua execução é comparado com o resultado esperado por ele. Através desta comparação é criado um *veredito*, que irá informar se o teste obteve falha ou sucesso.

O *oráculo de teste* é o mecanismo de uma ferramenta de teste responsável pela comparação dos resultados e criação dos vereditos.

As ações de um caso são chamadas de *ações de teste* e é através delas que os testes sobre um software são executados. As ações podem observar ou controlar o software que está sendo testado. Quando as ações de teste somente monitoram a execução do software, sem exercer qualquer influencia sobre a sua execução, então o teste é chamado de *teste de observação*. Existem casos em que se faz necessário o controle sobre o software que está sendo testado. Nesse caso, as ações de um caso de teste controlam o software, manipulando as suas entradas, operações e comportamentos.

O termo contaminação do Sistema Sob Teste (*System Under Test*, SUT) também é utilizado no decorrer desse trabalho. O SUT é descrito pela ISO/9646 *Conformance Testing Methodology and Framework* (CTMF) [2] como o sistema que está sendo testado. A contaminação do SUT é caracterizada pela inclusão de código no SUT [12] (i.e., inserção de código de teste no software que está sendo testado). Isso normalmente é utilizado para alterar alguma característica do SUT e adaptá-lo às ações de um caso de teste.

Ainda, Ammann e Offut [3] sugerem a adoção das nomenclaturas *testes estáticos* e *testes dinâmicos*. Os testes estáticos fazem a inspeção do software sem a sua execução, nor-

malmente através da análise sintática de seu código. Já os testes dinâmicos inspecionam o software enquanto o mesmo é executado.

3.2 Níveis de Teste de Software

Cada atividade do desenvolvimento de software possui um ou mais níveis de teste. Segundo Ammann e Offut [3], estes níveis são divididos em: teste de aceitação, teste de sistema, teste de integração, teste de módulo e teste unitário. Sendo os diferentes níveis de teste são ilustrados na Figura 3.2.

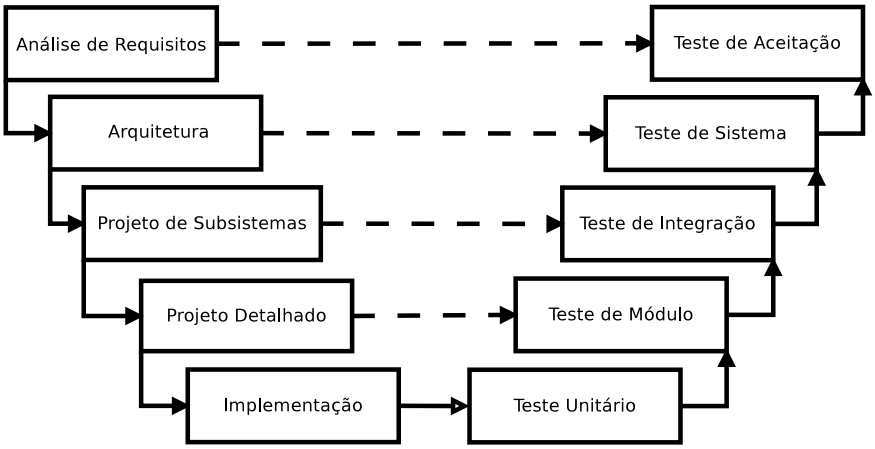


Figura 3.2: Níveis de teste.
Fonte: Figura adaptada de Amman e Offut [3].

Em um projeto de software, a fase de análise de requisitos descreve as necessidades e expectativas do usuário, então o *teste de aceitação* determina o quanto um software atende às expectativas do usuário. A arquitetura do software, por sua vez, visa integrar os componentes de um software com o ambiente sobre o qual eles irão executar. Para verificar se esse processo ocorre de forma correta utiliza-se o *teste de sistema*.

Na fase de projeto dos subsistemas o software é dividido em partes, cada qual com suas funcionalidades específicas, que são testadas individualmente. O *teste de integração* é utilizado para validar a integração dos subsistemas e verificar a interface entre eles. Este teste assume que os módulos funcionam corretamente, pois já deveriam ter sido testados no *teste de módulos*.

Um módulo é formado por várias unidades, no *teste de módulo* as integração destas unidades é testada para cada um dos módulos do sistema. As unidades são as menores

porções de um sistema (e.g., funções, *loopings*, laços condicionais) e são testadas através do *teste unitário*.

3.3 Teste de Sistemas Distribuídos

Testar sistemas distribuídos é uma tarefa difícil. Nesses sistemas, características de ambiente, como volatilidade (i.e., entrada e saída de elementos/nodos na rede), alto índice de falhas e elementos heterogêneos, normalmente estão presentes. As ferramentas para teste de sistemas distribuídos consideram que todos os nodos do sistema distribuído permanecem disponíveis durante a execução de um caso de teste. Se algum nodo entrar ou sair da rede ou falhar, enquanto um caso de teste é executado e a ferramenta não conseguir identificar esta volatilidade, então o resultado do teste pode ser comprometido. Computadores e equipamentos de rede heterogêneos também comprometem os testes de sistemas distribuídos, pois caso os componentes (i.e., nodos) do sistema distribuído não possuam as mesmas configurações, alguns podem se comunicar ou processar de uma maneira distinta aos demais. Por exemplo, se existir um nodo mais lento que os outros, ele pode gerar um atraso na comunicação ou no processamento. Para que a heterogeneidade não interfira nos testes de sistemas distribuídos, a ferramenta de teste precisa conhecer as características de cada nodo e gerenciar os testes baseando-se nestas características. Porém, é preciso um grande esforço para que as características de ambiente sejam tratadas e não interfiram no resultado dos testes.

Almeida [12] propõe três abordagens para testar sistemas distribuídos: um nodo representando todo o sistema, a execução de casos de teste em paralelo e a decomposição do caso de teste em ações. Em todas elas é utilizado um caso de teste simples. Este insere um valor no sistema distribuído e posteriormente o recupera, verificando se o resultado recuperado é o mesmo do inserido.

A primeira abordagem considera um único nodo de um sistema distribuído como se ele fosse todo o sistema. Ela é ilustrada na Figura 3.3.

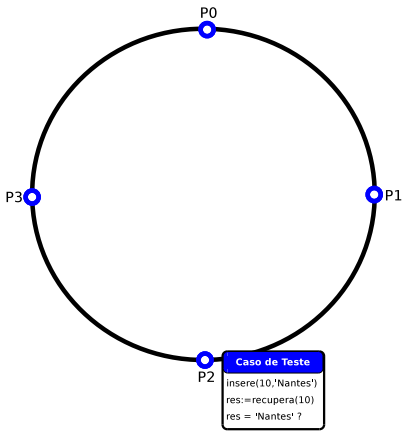


Figura 3.3: Primeira abordagem de teste de sistemas distribuídos.
Fonte: Figura adaptada de Almeida [12].

Nela, somente este nodo recebe o caso de teste a ser executado e caso o teste seja bem sucedido sobre esse nodo, então assume-se que ele será em todo o sistema distribuído. Mas nessa abordagem, a comunicação entre os nodos e outras características do sistema distribuído, como a volatilidade, por exemplo, não estão consideradas. Portanto, elas não são testadas, o que pode gerar resultados incompletos e pouco precisos. Isso porque, se o mesmo teste fosse executado de uma forma distribuída, ou seja, em mais do que um nodo do sistema distribuído, as características desse sistema, estariam presentes e elas poderiam modificar o resultado do teste.

A Figura 3.4 ilustra as segunda e a terceira abordagens propostas em [12].

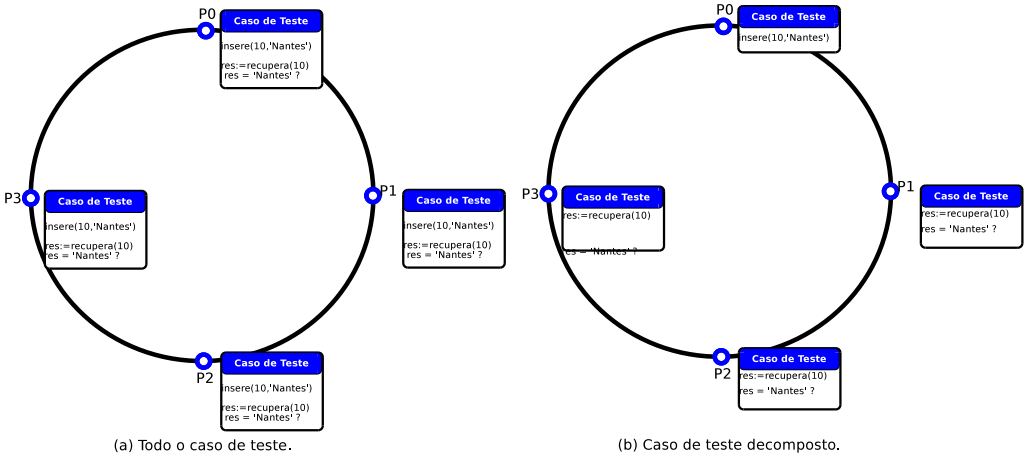


Figura 3.4: Segunda e terceira abordagens de teste de sistemas distribuídos.
Fonte: Figura adaptada de Almeida [12].

Na segunda abordagem, múltiplos nodos de um sistema distribuído recebem diferentes casos de teste. Dessa forma, cada um dos nodos executa paralelamente aos demais, o que

possibilita a execução de mais do que um caso ao mesmo tempo. Mas essa abordagem ainda não permite que um mesmo caso de teste seja executado de maneira distribuída.

Para que um mesmo caso de teste seja executado de forma distribuída, a terceira abordagem propõe um modelo de execução onde um caso de teste é dividido em ações de teste. Cada ação corresponde a uma ou mais interações do caso de teste com o SUT, como por exemplo, inserir ou recuperar um dado. Nessa abordagem, as ações são atribuídas a nodos distintos de um mesmo sistema distribuído. Dessa forma, o caso de teste é executado de uma maneira distribuída, considerando as características do sistema distribuído sobre o qual ele está sendo executado. Com isso, a terceira abordagem testa sistemas que executam em um ambiente com características que normalmente seriam encontrada em um ambiente real (i.e., ambiente onde o sistema executa em produção).

Na terceira abordagem, como as ações são executadas em diferentes nodos de um sistema distribuído, é necessário saber a sequência de execução delas, bem como, quando as ações iniciam e terminam. Caso contrário, considerando como exemplo o caso de teste utilizado nesta seção, a ação de recuperação poderia ser executada antes da ação de inserção do dado. Isso originaria um erro, já que o dado não teria sido inserido no sistema distribuído, e portanto, não seria recuperado. Para evitar isso é necessária uma ferramenta que distribua as ações de um caso de teste sobre um sistema distribuído e sincronize a execução delas. Algumas ferramentas com esse propósito são descritas na Seção 3.3.1.

3.3.1 Soluções para Teste de Sistemas Distribuídos

A arquitetura clássica para coordenação de testes em sistemas distribuídos consiste em um componente central (coordenador) e vários testadores. O coordenador envia as especificações de teste, através de um caso de teste, controla a sincronização das ações e recebe as saídas de cada ação de teste. Já os testadores executam as ações de teste contidas no caso de teste.

O Joshua [22] é uma das primeiras arquiteturas que utilizou coordenador centralizado. Sua arquitetura pode ser observada na Figura 3.5.

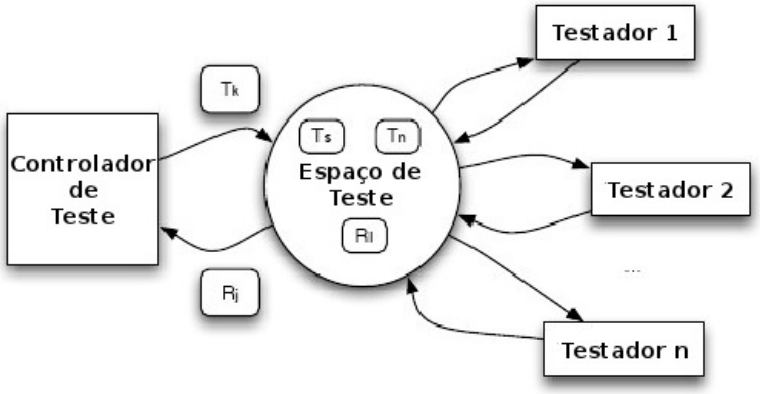


Figura 3.5: Arquitetura do Joshua.
Fonte: Figura adaptada de Kapfhammer [22].

Os testadores buscam pelos casos de teste, no espaço de teste, e após executá-los, gravam seus resultados novamente no espaço de teste. O Joshua é formado por um componente centralizado, chamado de controlador de teste, que recebe os casos de teste e os escreve no espaço de teste. O controlador monitora o espaço de teste para encontrar os resultados de cada testador e gerar um resultado final.

O GridUnit [7] é uma solução de teste que se baseia no Joshua. Seu objetivo principal é implantar e controlar testes unitários sobre grades computacionais. No GridUnit cada teste é transformado em uma tarefa para grade computacional e diferentes testes podem ser executados por componentes distintos de uma mesma da grade. O problema do GridUnit, é que ele, assim como o Joshua, não consegue dividir um caso de teste em ações, para que elas sejam executadas paralelamente.

Outra arquitetura, proposta por Parveen et al. [28], chamada de HadoopUnit, possibilita o paralelismo de casos de teste através do Hadoop [4] (i.e., uma implementação MapReduce de código aberto). A execução do HadoopUnit é similar ao GridUnit, porém troca a Grid pelo Hadoop e ele também não permite a decomposição do caso de teste em ações.

A abordagem de Rings et al. [30], chamada de Aplicação para Gestão de Testes (*Test Application Management*, TAM), consegue decompor um caso de teste em ações. O TAM se baseia na CTMF e sua linguagem de teste na *Testing and Test Control Notation* (TTCN-3) [19]. Devido à portabilidade da CTMF e da linguagem TTCN-3, essa

arquitetura também é portável para ambientes distribuídos distintos. No TAM é feita uma divisão da tarefa em pequenas sub-tarefas, executando-as sobre o ambiente de grade computacional. Após o término da execução de todas as sub-tarefas é feita a combinação dos seus resultados, agrupando o resultado de cada uma delas. O problema do TAM é que ele não mantém controle sobre o SUT, assim como as outras arquitetura até aqui descritas, o que em algumas situações é necessário (e.g., para simular características do SUT).

O Jata [39] é outra solução de teste que utiliza a linguagem TTCN-3. Ele testa os sistemas distribuídos através do JUnit [32] (i.e., framework para testes unitários na linguagem de progração Java). No Jata, os casos de teste são criados com a linguagem TTCN-3 e as ações são executadas pelo JUnit. Por utilizar a TTCN-3 como linguagem para seus casos de teste o Jata, assim como o TAM, é adaptável a diferentes tipos de sistemas distribuídos. O Jata também não consegue manter controle sobre o SUT e emula os componentes do sistema distribuído, o que pode contaminar o código do SUT.

O PeerUnit [12] é uma abordagem para teste de sistemas distribuídos que também divide seus casos de teste em ações. Ele permite testar ambientes distribuído com alta volatilidade e é altamente escalável (i.e., 4096 nodos). A arquitetura do PeerUnit é mostrada na Figura 3.6.

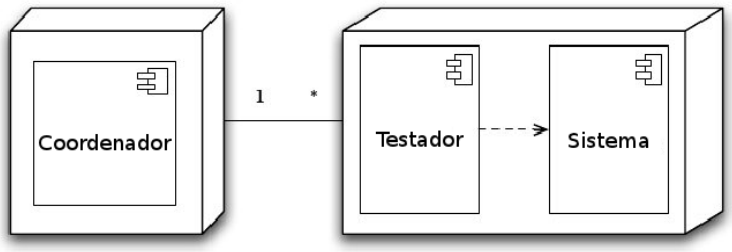


Figura 3.6: Diagrama de distribuição do PeerUnit.
Fonte: Figura adaptada de Almeida [12].

No PeerUnit o coordenador é executado separadamente e os testadores executam, cada qual em um nodo do sistema distribuído que está sendo testado. O coordenador é responsável por gerenciar os diversos testadores, dividir um caso de teste em ações e as atribuí-las aos testadores que devem executá-las. Os testadores controlam os nodos do sistema distribuído que estão testando e executam as ações que lhes são atribuídas pelo

coordenador.

Em Walter et al. [37] são utilizados os componentes chamados de caixa de ferramentas. As caixas de ferramentas podem ser combinadas de diversas maneiras, adaptando-se a diferentes tipos de ambientes a serem testados. É uma proposta conceitual de arquitetura de teste, que considera os conceitos da CTMF, não sendo possível implementá-la. Esta estrutura pode utilizar um coordenador centralizado (MTC) ou distribuído. No caso de não utilizar coordenador centralizado, então os próprios TCs devem conhecer um algoritmo comum entre eles para a sincronização das ações de teste. Por tratar-se de uma proposta conceitual, ela não pode ser comparada com as demais.

Ulrich e Konig [35] apresentam uma arquitetura para teste sem coordenador centralizado. Esta é chamada de *Test and Monitoring Tool* (TMT). O TMT é formado por vários testadores, cada qual sendo responsável por um *Partial Test Case* (PTC). Vários PTCs podem ser executados em paralelo e cada um é responsável por testar uma porção de um caso de teste. Todos os PTCs comunicam-se entre si, escalonando as ações e evitando que um PTC interfira na execução de outro. No TMT não há a necessidade de um coordenador centralizado. O TMT não trata de forma correta a volatilidade dos nodos da rede, podendo causar *deadlocks* no processo de sincronização.

Em Singh et al. [31] é proposto um *overlay*, através do P2 System [1]. Faz uso de uma linguagem Overlog, que é uma linguagem declarativa de alto nível. Esta é traduzida em um grafo, que pode ser executado, sendo que todas as operações de teste baseiam-se nesse grafo. Essa abordagem simula a estrutura da rede baseando-se nos conceitos de cada tipo de sistema distribuído, não implementando um ambiente real.Além disso, o P2 System somente avalia o ambiente distribuído, mas não testa os sistemas que são executados sobre o ambiente.

Existem outras soluções para teste de sistemas distribuídos que não utilizam um coordenador central, entre elas o P2P tester [9], o Pigeon [40] e o framework apresentado em Hughes et al. [21]. Estas soluções realizam os testes analisando os arquivos de log, após a execução do caso de teste. Elas não testam os sistemas em tempo real, portanto elas não consideram as características dos sistemas distribuídos (e.g., comunicação entre os nodos,

atrasos de rede), o que pode gerar resultados pouco precisos.

3.3.2 Comparativo das Soluções para Teste de Sistemas Distribuídos

As soluções para teste de sistemas distribuídos são comparadas nesta seção. No comparativo são considerados o controle sobre o SUT, o tipo de execução do teste, a decomposição do caso de teste em ações e a não contaminação do SUT.

Às vezes, quando se testa sistemas distribuídos é necessário controlar o SUT. Isso é feito para simular falhas ou algum outro tipo de comportamento no sistema que está sendo testado. Também é importante que a ferramenta de teste consiga executar testes dinâmicos. Assim, ela testa o sistema durante a sua execução, dessa forma, as características do sistema distribuído, como atrasos na rede, volatilidade, entre outras, são consideradas durante o teste. Além delas, a decomposição do caso de teste em ações também é uma característica necessária para testar sistemas distribuídos, pois possibilita a execução de operações em nós distintos do sistema distribuído. Muitas soluções de teste precisam alterar o código do SUT para conseguir controlá-lo e sempre que isso é feito, existe a possibilidade de algum código com erro ser adicionado ao SUT. A adição de código no SUT é chamada de contaminação do SUT, pois o código adicional pode modificar suas características originais e gerar erros de execução. Então se o SUT apresentar algum erro durante a execução dos testes é difícil identificar se o erro foi ocasionado por uma falha do SUT ou pelo código que foi inserido nele. Portanto, a não contaminação do SUT também é uma característica importante no teste de sistemas distribuídos.

Considerando as características requeridas, na Tabela 3.3.2 é mostrado o comparativo das soluções para teste de sistemas distribuídos.

Tabela 3.1: Comparativo das soluções para teste de sistemas distribuídos.
Fonte: Tabela adaptada de Almeida [12].

Arquitetura	**Controle do SUT**	**Tipo de Execução**	**Decomposição em ações**	**Contaminação do SUT**
Joshua	Não	Dinâmica	Não	Não
GridUnit	Não	Dinâmica	Não	Não
HadoopUnit	Não	Dinâmica	Não	Não
TAM	Não	Dinâmica	Sim	Sim
Jata	Não	Dinâmica	Sim	Sim
Singh et al. [31]	Sim	Sim	Sim	Sim
TMT	Não	Dinâmica	Sim	Sim
Pigeon	Não	Dinâmica	Não	Sim
P2PTester	Não	Dinâmica	Não	Sim
Hughes et al. [21]	Não	Dinâmica	Não	Sim
PeerUnit	Sim	Dinâmica	Sim	Não

Analisando o comparativo feito na Tabela 3.3.2 somente a solução de Singh et al. [31] e o PeerUnit mantém controle sobre o SUT. Todas as soluções são capazes de executar testes dinâmicos e somente as soluções TAM, TMT, e o PeerUnit decompõem os seus casos de teste em ações. Quanto à contaminação do SUT, o Joshua, o GridUnit, o HadoopUnit e o PeerUnit não o fazem. Com base nos dados comparativos apresentados nessa tabela, o PeerUnit é a solução de teste que mais se adapta para testar sistemas distribuídos.

3.4 Teste de Sistemas Baseados em MapReduce

A adoção do MapReduce vem crescendo em muitas organizações [10]. Para que a qualidade dos sistemas para MR seja garantida, eles devem ser testados. Então, nesse capítulo são comparadas algumas soluções para teste de sistemas baseados em MapReduce.

3.4.1 Soluções para Teste de Sistemas Baseados em MapReduce

As soluções para teste de sistemas baseados em MapReduce apresentadas nesta seção são: o MRUnit (Seção 3.4.1.1), a solução de Venner [36] (Seção 3.4.1.2), o Ganesha (Seção 3.4.1.3) e o Herriot (Seção 3.4.1.5).

3.4.1.1 MRUnit

A primeira solução de teste a ser descrita é o MRUnit. O MRUnit é uma solução para teste de sistemas baseados em MapReduce proposta em Cloudera [10]. Ela faz uso do JUnit [32], testando a entrada e saída de cada tarefa. É disponibilizado juntamente com a distribuição Hadoop da Cloudera. Ela restringe-se a testes unitários e não mantém controle sobre o SUT, além de contaminar o código do SUT, o que pode gerar vereditos falso-positivos.

3.4.1.2 Solução de Venner

Outra solução de teste é proposta em Venner [36]. Esta realiza testes unitários e de depuração através da classe *org.apache.hadoop.mapred.ClusterMapReduceTestCase*, disponibilizada juntamente com o Hadoop [4]. Nesta solução, todos os componentes do MapReduce são executados na mesma Máquina Virtual Java (JVM) e o *Java Plataform Debugger Architecture* (JPDA) [25] é utilizado para conectar à JVM e executar tarefas de depuração e de monitoramento. Com isso, é possível conectar, testar e monitorar cada componente do MapReduce. O problema desta solução é que ela não contempla a execução distribuída do MapReduce, somente local (i.e., em uma mesma JVM) e ela não mantém controle sobre o SUT.

3.4.1.3 Ganesha

A ferramenta Ganesha [27] analisa a execução dos sistemas baseados em MapReduce através do SALSA [26] (i.e., uma ferramenta que analisa o fluxo de execução de sistemas através da análise de arquivos de log). O Ganesha gera o resultado de seus testes baseando-se em hipóteses, sendo que uma anormalidade nas hipóteses caracteriza-se como um provável ponto de falha. As métricas para composição das hipóteses são colhidas através do programa *sysstat* (distribuído juntamente com alguns sistemas operacionais), que é executado a cada segundo. Através do *sysstat*, para cada componente, são coletados vetores, que armazenam informações de seus comportamentos (perfis).

A execução do Ganesha divide-se em duas fases: treinamento e teste. Na fase de treinamento, acontece uma espécie de aprendizagem, através dos vetores coletados em um ambiente livre de falhas. Para isso, são submetidos K trabalhos para que o MapReduce execute-os e em um certo intervalo de tempo é feita uma busca pelo comportamento dos componentes. O comportamento é chamado de *perfil* e é utilizado como base para as técnicas de aprendizagem. Na fase de teste, é verificado se existe algum erro no sistema MapReduce que está sendo executado. Isso pode ser feito, por exemplo, comparando a saída do sistema com o resultado esperado. Caso algum erro seja encontrado, então é feita uma busca pelo componente do MR que causou esse problema.

A busca pelo componente causador do erro baseia-se na similaridade dos componentes durante a execução do teste. Ou seja, os componentes devem apresentar um comportamento similar durante o período de tempo em que estão executando. Para cada componente o Ganesha mantêm um histograma baseado no perfil daquele componente. Quando o Ganesha recebe uma novo perfil, referente àquele componente, é feita uma comparação com os perfis anteriores. É esperado que o histograma dos perfis de cada componente mantenha-se coerente durante o período de execução. Se um histograma apresentar um comportamento diferenciado, então o componente referente a ele é considerado como o componente causador do erro.

O *sysstat* busca por métricas de sistema operacional e de rede. Portanto, as métricas utilizadas pelo Ganesha não correspondem exatamente aos componentes do MapReduce e sim, ao sistema operacional sobre o qual estes componentes estão sendo executados. Durante a execução do Ganesha outros sistemas podem estar executando concorrentemente. Estes sistemas tendem a interferir nos perfis colhidos pelo *sysstat*, já que os recursos de rede e o sistema operacional são disputados pelos componentes do MapReduce e pelos sistemas concorrentes. Dessa forma, o resultado dos testes do Ganesha não são confiáveis, já que baseiam-se em métricas que podem sofrer influências externas, como por exemplo, dos sistemas concorrentes.

O Ganesha é uma solução que não controla os componentes do MapReduce, ele os testa de forma passiva (i.e., sem interação). Para verificar se um sistema para MapReduce foi

executado corretamente ou com falha, o Ganesha somente analisa os arquivos de log deste sistema, após ele ter sido executado. Então, no Ganesha, os testes não são executados de forma dinâmica (i.e., durante a execução do SUT) e dessa forma, eles desconsideram as características de execução do sistema que está sendo testado.

3.4.1.4 Fail Testing Service (FTS)

O *Fail Testing Service* (FTS) [20] é utilizado para depurar erros no MapReduce. Ele cria um cenário de teste, onde são injetadas algumas falhas e através delas são identificados pontos de falha e listas dos possíveis erros que estas falhas podem gerar. O FTS faz uso da linguagem *Declarative Testing Specification* (DTS), o que possibilita que o código de teste seja especificado em poucas linhas.

O FTS cria pontos de depuração sobre o cenário de teste. Para definir um ponto de depuração criam-se os **IDs de falha**, que são compostos pelo ponto de falha e pela falha a ser injetada naquele ponto. Após a identificação dos IDs de falha o FTS gera uma lista das possíveis falhas que podem ocorrer para cada um deles, considerando o ponto de falha e a falha a ser injetada. Para cada ponto de falha podem ser gerados mais do que um ID de falha, somente variando a falha injetada.

O FTS primeiramente instancia o SUT e sobre ele insere uma superfície de falhas (i.e., responsável por criar os IDs de falha). Os IDs de falha, criados na superfície de falha, então são enviados a um servidor FTS, que mantém uma lista contendo todos os IDs de falha. Após a superfície de falhas ser criada e os IDs de falha enviados ao servidor, então o ambiente do FTS já está pronto para receber os testes.

Antes dos testes são injetadas as falhas correspondentes aos pontos de falha, especificadas nos IDs de falha. Alguns sistemas então são executados sobre a superfície de testes do FTS, após a injeção de cada uma das falhas, verificando se com aquela falha eles executam corretamente. Os possíveis erros gerados pela injeção de falhas são armazenados em uma lista de erros, que contém o ID de falha (ponto de falha e falha injetada) e o erro gerado por aquele ID de falha. Então, se futuramente algum teste encontre um dos erros inseridos na lista de erros, o FTS sugere que o erro foi ocasionado pelo ID de falha

correspondente ao erro, não sendo necessário depurar o sistema.

O FTS gera conclusões baseadas em hipóteses, pois busca por erros em uma lista criada através de execuções anteriores. Isso pode gerar resultados falso-positivos, pois talvez um mesmo erro seja gerado por falhas e pontos de falha diferentes. Ele somente permite que sejam executados testes de depuração e por isso, não se adéqua ao propósito deste trabalho, que é executar diferentes tipos de testes dinâmicos.

3.4.1.5 Herriot

O Herriot [5] é uma solução de testes mantida pela equipe do Hadoop [4], que utiliza o JUnit para a implementação dos testes. A arquitetura do Herriot é mostrada na Figura 3.7.

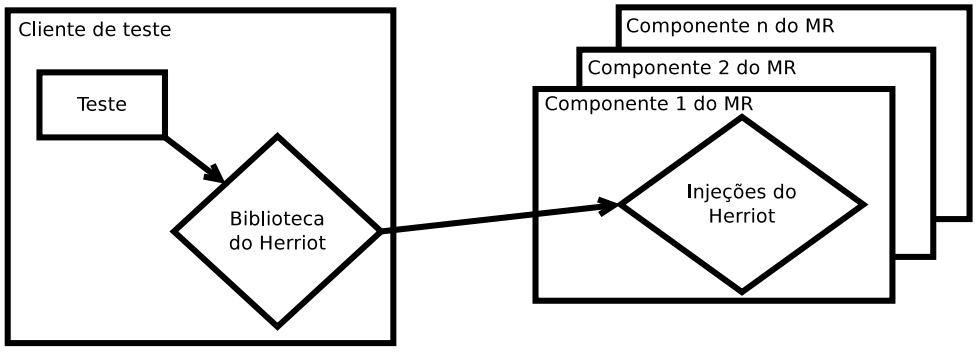


Figura 3.7: Arquitetura do Herriot.
Fonte: Figura adaptada de Herriot [5].

Conforme a figura, no cliente de teste fica localizada a biblioteca do Herriot e todo teste é intermediado por ela, sendo ela quem injeta as ações de teste nos componentes do MapReduce. O Herriot não mantém o controle sobre os componentes do MapReduce durante a execução de seus teste. Ele é uma solução distribuída juntamente com o Hadoop e não é capaz de testar outras implementação de MapReduce.

3.4.2 Comparativo das Soluções para Teste de Sistemas Baseados em MapReduce

No comparativo das soluções para testar sistemas baseados em MapReduce são consideradas as mesmas características utilizadas na Seção 3.3.2. O comparativo destas soluções é mostrado na Tabela 3.4.2.

Tabela 3.3: Comparativo das soluções para teste de sistemas baseados em MapReduce.

Arquitetura	**Controle do Ambiente**	**Execução do Teste**	**Decomposição em ações**	**Contaminação do SUT**
MRUnit	Não	Dinâmica	Sim	Não
Herriot	Não	Dinâmica	Não	Não
FTS	Não	Dinâmica	Não	Sim
Ganesha	Não	Dinâmica	Não	Não
Venner [36]	Não	Dinâmica	Não	Não

Nenhuma das soluções mostradas na tabela satisfaz todos os requisitos para testar sistemas baseados em MapReduce (i.e., controle sobre o SUT, execução de testes dinâmicos, decomposição do caso de teste em ações e não contaminação do SUT). Das ferramentas comparadas, nenhuma delas mantém controle sobre o ambiente, somente o MRUnit é capaz de decompor seus casos de teste em ações e o FTS contamina o SUT. Com base neste comparativo, nenhuma das soluções descritas na Seção 3.4.1 possuem as características necessárias para testar sistemas baseados em MapReduce de forma eficiente. Se forem comparadas também as ferramentas para teste de sistemas distribuídos (Seção 3.3.1), pode ser observado que o PeerUnit é a única solução de teste que atende a todos os requisitos para testar sistemas baseados em MapReduce. Porém, o PeerUnit é uma solução para teste de sistemas Peer-to-Peer (P2P), não estando apta a testar sistemas para MapReduce. Então, para suprir essa necessidade, no Capítulo 4 é mostrado o HadoopTest, um controlador de testes para sistemas baseados em MapReduce, que atende aos requisitos comparados nesta seção.

CAPÍTULO 4

HADOOPTEST: UM CONTROLADOR DE TESTES DE SISTEMAS BASEADOS EM MAPREDUCE

Neste capítulo é descrita a arquitetura do HadoopTest, nosso controlador de testes para sistemas baseados em MapReduce (MR). Ele é baseado no controle individual dos componentes MR que normalmente estão distribuídos sobre clusteres de computadores. Nossa solução permite que sejam criados e implantados casos de teste para testar as implementações do MR e as aplicações que são executadas sobre elas. Estes casos de teste podem combinar injeção de falhas com testes funcionais, além de o HadoopTest possibilitar que as ações do caso de teste sejam executadas paralelamente. Os casos de teste para o HadoopTest não são intrusivos, ou seja, não é alterado o código fonte do SUT. O HadoopTeste pode inicializar os componentes do MapReduce, controlá-los, monitorá-los e injetar falhas neles.

4.1 Definições do HadoopTest

Nesta seção definimos as principais características e nomenclaturas do HadoopTest. O caso de teste global, escalonamento, ações e vereditos locais seguem as definições do PeerUnit, descritas em Almeida [13].

Definimos C como o conjunto dos componentes da implementação MapReduce que irão ser testados. É necessário que os mesmos sejam identificados, pois eles são responsáveis por diferentes atividades e possuem interfaces distintas dentro de uma implementação MR. Sendo assim, componentes MR distintos poderão receber ações de teste distintas.

Definição 4.1.1 (Componentes do MapReduce) *O conjunto de componentes, chamados de C, são a junção do componente principal (mestre), chamado de M, com os componentes secundários (MR workers), chamados de W, onde $W = \{w_0, ..., w_n\}$.*

Para controlar e monitorar os componentes do MapReduce criamos os testadores. Simbolizamos com T o conjunto de testadores que controlam o conjunto de componentes MR, sendo que existe exatamente um testador para cada componente do MR (i.e., $|T| = |C|$).

Para testar os componentes do MapReduce deve ser criado um caso de teste global τ. τ interage com o SUT, sendo que ele possui um conjunto de ações A, que são executadas sobre C.

Definição 4.1.2 (Caso de Teste Global) *Segundo Almeida [12], um caso de teste global é formado por uma tupla $\tau = (A^\tau, T^\tau, L^\tau, S^\tau)$, onde:*

- *A^τ é o conjunto ordenado de ações $\{a_0^\tau, \ldots, a_n^\tau\}$, onde $A^\tau \subseteq A$;*

- *T^τ é o conjunto de testadores, onde $T^\tau \subseteq T$;*

- *L^τ é o conjunto de vereditos locais;*

- *S^τ é o escalonador.*

Cada ação correspondem a estímulos no SUT, que serão executados por testadores presentes em T sobre os componentes de C que eles controlam.

Definição 4.1.3 (Ações de teste) *A definição ação de teste proposta por Almeida [12] foi estendida neste trabalho, resultando em uma tupla $a_i^\tau = (h, I, \iota, T')$, onde:*

- *a_i^τ corresponte a uma ação, com identificador i, que faz parte do caso de teste τ;*

- *h é o nível hierárquico da ação;*

- *I é o conjunto de instruções da ação;*

- *T' é o conjunto de testadores responsáveis por executarem a ação, onde $T' \subseteq T$;*

- *ι é o intervalo de tempo máximo que a ação pode ser executada.*

As ações possuem um nível de hierarquia, definida pelo parâmetro h, que corresponde à sequência de execução da ação no caso de teste. Naturalmente, as ações que possuem o mesmo nível hierárquico (i.e., h) são executadas paralelamente. Cada ação possui um conjunto de instruções, sendo que as instruções tipicamente são chamadas às interfaces dos componentes do sistema, ou métodos da própria linguagem de programação e são elas que realizam os testes de cada ação. Em sistemas distribuídos, a autonomia e heterogeneidade dos nós interfere diretamente na execução dos serviços requisitados. Enquanto nós próximos conseguem responder rapidamente, nós distantes ou nós com *overload* podem demorar para responder. Isso pode retardar e/ou comprometer a execução dos casos de teste. Para evitar que as ações do caso de teste esperem por um tempo desnecessário, é atribuído a elas um tempo de execução máximo (i.e., ι).

Cada ação está associada a um conjunto de testadores que irão executá-la. É o escalonador quem faz o mapeamento entre as ações e o conjunto de testadores, verificando o parâmetro T' de cada ação.

Definição 4.1.4 (Escalonador) *Segundo Almeida [12], o escalonador é um mapeamento* $S = A \mapsto \Pi$*, onde* Π *é uma coleção de testadores (i.e.,* $\Pi = \{T_0, \ldots, T_n\}$*), onde* $T_i \subseteq T$*.*

Para definir se o resultado de uma ação foi correto ou não, cada testador gera um veredito para cada uma das ações que ele executa, que é chamado de veredito local.

Definição 4.1.5 (Veredito local) *Conforme Almeida [12] um veredito local é determinado pela comparação do resultado esperado, chamado de* E*, com a saída, chamada de* O*. O veredito local* l *de* τ*, com tempo de execução máximo* ι*, corresponde ao resultado de uma ação e é definido da seguinte maneira:*

$$l_\iota^\tau = \begin{cases} pass, & se\ O = E \\ fail, & se\ O \neq E \\ inconclusive, & se\ O = \emptyset \end{cases}$$

4.1.1 Caso de Teste de Exemplo

Para ilustrar melhor as definições da Seção 4.1, aqui é descrito um caso de teste de exemplo (Tabela 4.1), o qual também será utilizado como exemplo nas seções posteriores. O objetivo é testar se o MapReduce é tolerante a falhas, para isso o caso de teste realiza as seguintes ações: inicializa os componentes principais e secundários do MapReduce; após os componentes terem sido inicializados, submete um trabalho para ser processado pelo MR; enquanto o trabalho está sendo executado pelo MapReduce, então um dos testadores do HadoopTest simula uma falha no componente que ele inicializou, fazendo com que este componente pare de executar; então é validado o resultado final do trabalho submetido ao MapReduce, testando se ele foi equivalente ao resultado esperado.

Tabela 4.1: Caso de teste de exemplo para MapReduce.

Ação	**Hierarquia**	**Testador**	**Nome da ação**	**Dependência**	**Timeout**
a_0	1	t_0	$inicializarM$	$-$	100
a_1	2	t_1-t_5	$inicializarW$	a_0	1000
a_2	3	t_0	$enviarTarefa$	a_1	1000000
a_3	3	t_2	$matarW$	a_1	1000
a_4	4	t_0	$verificarResultado$	a_2	10000
a_5	5	t_1, t_3-t_5	$pararW$	a_1	1000
a_6	6	t_0	$pararM$	a_0	1000

Este caso de teste utiliza seis testadores $T^{\tau} = \{t_0, \ldots, t_5\}$, onde o testador t_0 inicializa o componente principal do MapReduce M e os demais testadores $\{t_1, \ldots, t_5\}$, os quais para efeito de descrição chamaremos de conjunto T^s, inicializam um componente secundário ($w \in W$) cada um, resultando em um testador para cada componente do MapReduce, onde $t_0 \rightarrow M$ e $(\forall t_i \rightarrow |W|)|i > 0)$, sendo que $\{t_0, \ldots, t_5\} \subseteq T \wedge M \subseteq C \wedge W \subseteq C$. Também foram utilizadas seis ações, sendo que as ações $inicializarM$ e $inicializarW$, respectivamente, inicializam o componente principal, pelo testador t_0, e os componentes secundários do MapReduce, pelos testadores t_1 a t_5. A ação $enviarTarefa$ envia um trabalho de para o MR, a ação $matarW$ remove um componente secundário e enquanto $enviarTarefa$ está executando, a ação $verificarResultado$ valida se a saída do trabalho corresponde com o resultado esperado. Por último, todos os componentes do MapReduce são parados, através das ações $pararW$ e $pararM$. Para cada ação foi estipulado um tempo máximo de execução (ι) e sua dependência, sendo que a ação $inicializarM$

não depende de nenhuma ação anterior para que possa ser executada e todas as demais possuem dependência, como a ação $inicializarW$, por exemplo, que depende da ação $inicializarM$.

4.2 Arquitetura do HadoopTest

O HadoopTest estende o *framework* PeerUnit [13]. Ele adiciona novas características ao PeerUnit, como por exemplo, dependência e execução paralela de ações, além de criar a interface de interação com o MapReduce. Sua arquitetura divide-se em dois componentes principais, o *coordenador* e o *testador*, sendo os dois ilustrados na Figura 4.1. O coordenador é responsável por gerenciar os testadores, além de distribuir um caso de teste sobre estes testadores e sincronizar a execução das ações do caso de teste. O testador é o componente do HadoopTest que interage com os nós do MapReduce e é através dele que as ações de um caso de teste são executadas.

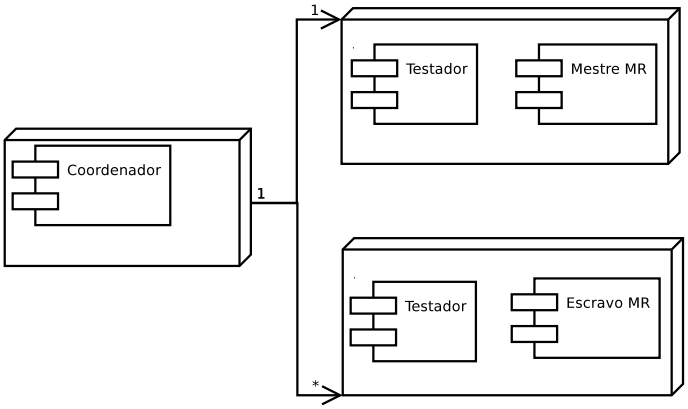


Figura 4.1: Componentes principais do HadoopTest.

Na Figura 4.1 pode-se observar que existe somente um coordenador para n testadores, sendo que um dos testadores inicializa o componente principal do MapReduce (i.e., M) e os demais testadores inicializam os componentes secundários do MR (i.e., W). A comunicação entre coordenador e testadores acontece utilizando Invocação Remota de Métodos (*Remote Method Invocation*, RMI) Java, o que permite a invocação de métodos em máquinas virtuais Java remotas. Os testadores recebem mensagens de coordenação do coordenador para executarem as ações de um caso de teste. As ações de teste são estímulos do HadoopTest nos componentes da implementação MapReduce, podendo es-

timular algum comportamento neles ou somente monitorá-los. As ações de teste podem ser utilizadas para testar tanto as implementações MapReduce, como as aplicações executadas sobre elas.

4.2.1 Aplicação do Caso de Teste de Exemplo utilizando o HadoopTest

Para explicar melhor o funcionamento do HadoopTest, nesta sessão mostramos a aplicação do caso de teste de exemplo (Tabela 4.1) sobre uma implementação do MapReduce. A Figura 4.2 mostra a estrutura de execução do MapReduce sendo controlada por testadores do HadoopTest.

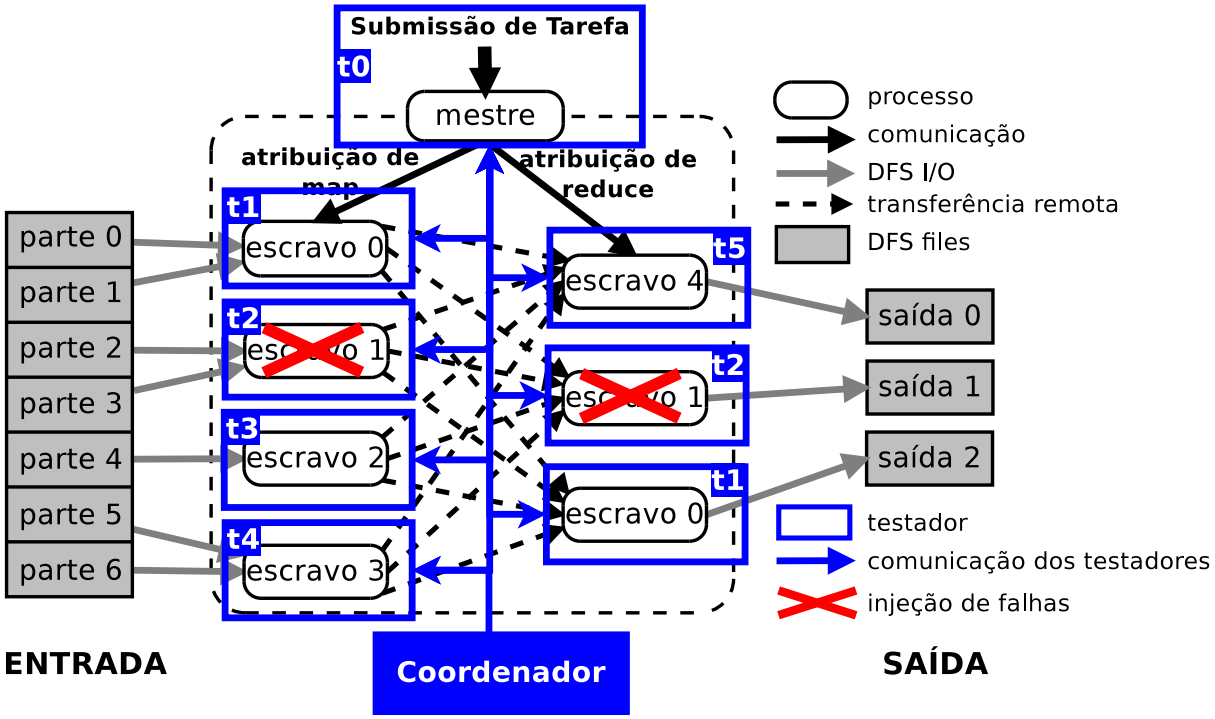


Figura 4.2: HadoopTest controlando uma implementação do MapReduce.

Nela existe um coordenador e seis testadores, identificados por $t_0 - t_5$. O coordenador não está associado a nenhum componente do MapReduce, pois como dito na Seção 4.2 ele somente é responsável por gerenciar os testadores e sincronizar a execução das ações do caso de teste, não interagindo com o SUT. Os testadores, por sua vez, inicializam, controlam e monitoram os componentes do MapReduce. Sendo que o testador t_0 controla o componente principal e cada um dos demais controla um componente secundário.

Neste cenário, cada testador do HadoopTest inicializa um dos componentes do MapRe-

duce e, a partir disso, o HadoopTest fornece interfaces para que seja mantido o controle sobre ele, podendo monitorá-lo e manipular o seu comportamento. Conforme o caso de teste mostrado na Tabela 4.1, a ação a_3 simula um erro no componente do MapReduce, que é controlado pelo testador t_2.

4.2.2 Coordenação da Execução das Ações de um Caso de Teste

No HadoopTest as ações de um caso de teste são executadas de forma sequencial. Sendo que cada ação possui um nível hierárquico e sua sequência de execução é condicionada ele. É o coordenador quem sincroniza a execução das ações, baseando-se no nível hierárquico delas. As ações com menor nível hierárquico são executadas antes do que as ações com maior nível. O processo de coordenação das ações de um caso de teste, através do HadoopTest, pode ser visualizado no diagrama de sequência ilustrado na Figura 4.3. Nele é mostrada somente parte da execução do caso de teste de exemplo, descrito na Tabela 4.1.

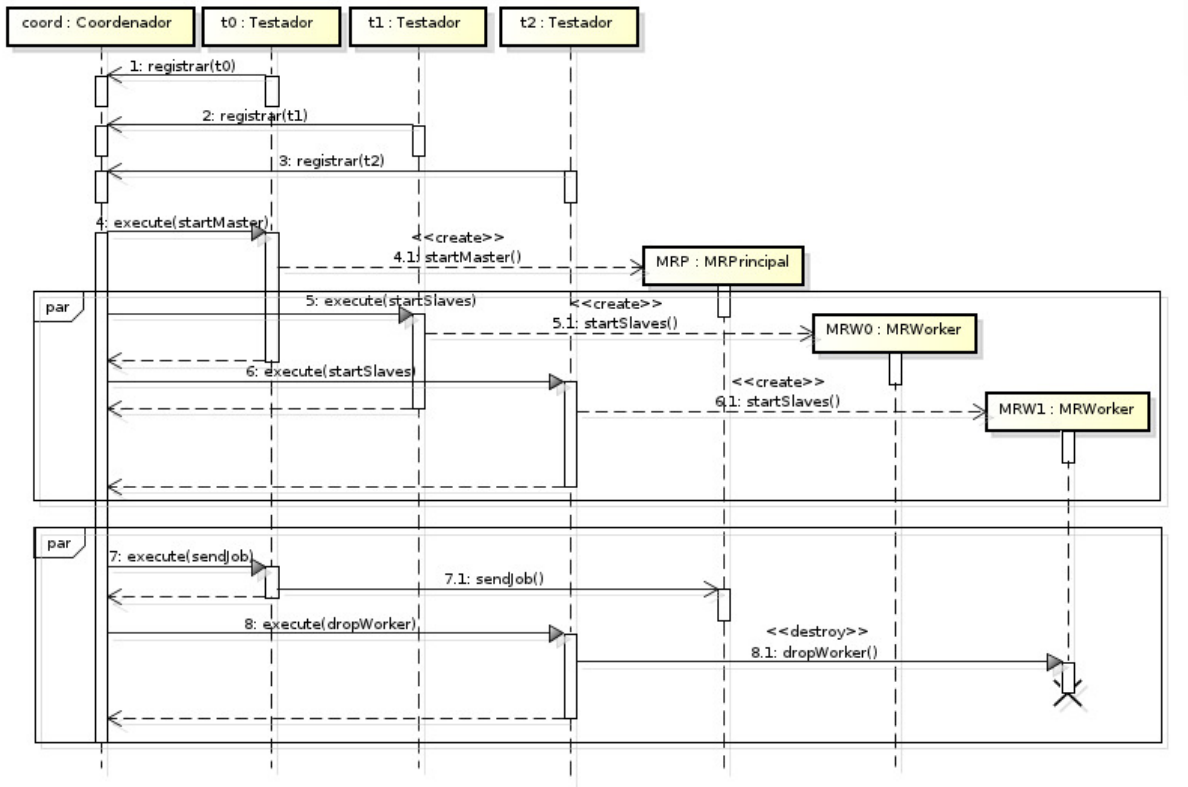


Figura 4.3: Diagrada de sequência de execução das ações do HadoopTest.

Após todos os testadores registrarem-se no coordenador do HadoopTest, então ele

inicia o gerenciamento de execução das ações do caso de teste. O coordenador solicita a todos os testadores responsáveis pela primeira ação do caso para que eles a executem. Assim que estes testadores terminam de executá-la, eles enviam seus vereditos locais desta ação para o coordenador, de forma assíncrona aos demais. Quando o coordenador recebe o veredito local de todos os testadores responsáveis pela ação, então ele a considera como concluída, e com isso, ele passa ao próximo nível hierárquico, assim se repetindo até que o último nível hierárquico seja atingido. Caso duas ou mais ações possuam níveis hierárquicos similares, como é o caso das ações *sendJob* e *dropWorker* do caso de teste mostrado na Figura 4.3, então elas são executadas em paralelo. Nesse caso, o coordenador espera até que ambas sejam concluídas por todos os testadores responsáveis por elas antes de passar para o próximo nível de execução.

4.2.2.1 Algoritmo para Sincronização da Execução das Ações de Teste

A execução dos testes através do HadoopTest é feita em três etapas, seguindo o modelo proposto por Almeida [12]: registro dos testadores, execução das ações dos casos de teste e criação dos vereditos dos casos de teste. Cada testador é responsável por executar um conjunto de ações de teste (i.e., A^t) e para que o coordenador consiga sincronizar a execução de todas as ações de teste, cada testador precisa se registrar no coordenador e informar seu conjunto de ações. O registro dos testadores e suas ações de teste é mostrado no Algoritmo 1, descrito em Almeida [12].

Algoritmo 1: Algoritmo para registro dos testadores no coordenador do HadoopTest.

Input: t;
Output: id;
foreach $a \in A^t$ **do**
  $S^\tau(a) \leftarrow S^\tau(a) \cup t$;
end
id++;
return id;

O coordenador identifica cada um dos testadores numericamente, inciando em 0 e sendo incrementado por 1 a cada testador que se registra. Após todos os testadores

se registrarem é criado um escalonador (i.e., S^τ), que armazena quais são os testadores responsáveis por cada uma das ações de teste. Após o registro de todos os testadores é sincronizada a execução das ações teste, conforme o Algoritmo 2.

Algoritmo 2: Algoritmo para sincronização da execução das ações de teste.

```
Input: A^τ- conjunto ordenado de ações de teste; T^τ- conjunto de testadores;
Output: veredito global;
S^τ ← ∅;
V^τ ← ∅;
forall t ∈ T^τ do
    S^τ ← register(t, A^τ);
forall a ∈ A^τ com o mesmo h do
    Coordenador envia mensagem para todos t ∈ T' ∧ t ↦ a;
    Coordenador espera mensagem de término de todos os testadores;
    if θ then
        V^τ ← V^τ ∪ {inconclusive};
    else
        V^τ ← V^τ ∪ t.result(a^h);
    if V^τ ⊆ {fail} then
        break;
return oracle(V^τ) ;
```

Após a etapa de registro, as ações de teste são executadas conforme seu nível hierárquico (h), do menor para o maior nível. Para que a execução das ações seja iniciada, o coordenador envia uma mensagem para os testadores responsáveis pelas ações do primeiro nível hierárquico ($A^\tau(h)|h = 1$), para que eles as executem. Se existir mais do que uma ação com o mesmo nível hierárquico, então elas são executadas paralelamente. Antes de cada uma das ações de teste ser executada é verificado se alguma ação anterior retornou erro (i.e., veredito *fail*). Caso qualquer uma das ações de teste recebeu o veredito *fail*, então todas as demais não são executadas. As ações de teste com nível hierárquico maior do que uma ação com erro (i.e., executadas após a ação com erro) não são executadas, pois seu veredito pode ser modificado devido ao erro da ação anterior.

Cada um dos testadores, assim que termina de executar as suas ações de teste, envia uma mensagem para o coordenador de forma assíncrona aos demais, informando o seu veredito para cada uma das ações executadas por ele. O coordenador identifica quais são os testadores que já terminaram de executar uma ação através dos vereditos que recebe. Dessa forma, quando todos os testadores responsáveis por uma ação de teste responderem para o coordenador, ele identifica que ela foi executada. Assim que todas as ações de teste

são executadas, então o coordenador gera um veredito final, através da função de oráculo definida por Almeida [12].

O HadoopTest introduz duas melhorias significativas na sincronização de ações descrita em Almeida [12]: a dependência de ações e a execução paralela de ações de teste. Através do HadoopTest, antes de cada uma das ações de teste ser executadas é verificada sua relação de dependência com ações anteriores. As ações de teste posteriores a uma ação com erro não são executadas, o que evita que elas sejam executadas desnecessariamente e aumenta a confiabilidade do resultado destas ações, já que seus vereditos poderiam ser alterados devido ao erro da ação anterior. Já a execução paralela de ações de mesmo nível hierárquico diminui o tempo de execução dos casos de teste e permite que operações distintas sejam executadas ao mesmo tempo (e.g., a simulação de falha em um componente enquanto outro componente executa alguma operação no ambiente distribuído).

4.3 Implementação do HadoopTest

Esta seção apresenta o diagrama de classes do HadooopTest e descreve a sintaxe para a criação de seus casos de teste. Ela está organizada da seguinte maneira: Seção 4.3.1, mostra o diagrama de classes do HadoopTest; Seção 4.3.2, descreve a sintaxe para criação de casos de teste para o HadoopTest.

4.3.1 Classes Principais

A Figura 4.4 ilustra as principais classes do HadoopTest e seus relacionamentos. Todas as classes foram implementadas através da linguagem de progração Java (versão 1.6). Estas classes estão relacionadas à interface de controle e interação do HadoopTest com o MapReduce, não atendo-se à interação entre coordenador e testadores, descrita no capítulo anterior. Na figura destacam-se as classes *MRAbstract* (que implementa as funcionalidades do HadoopTest), *TestMRApplication* (que corresponde ao Caso de Teste) e *MRApplication* (relacionada à aplicação MR a ser testada).

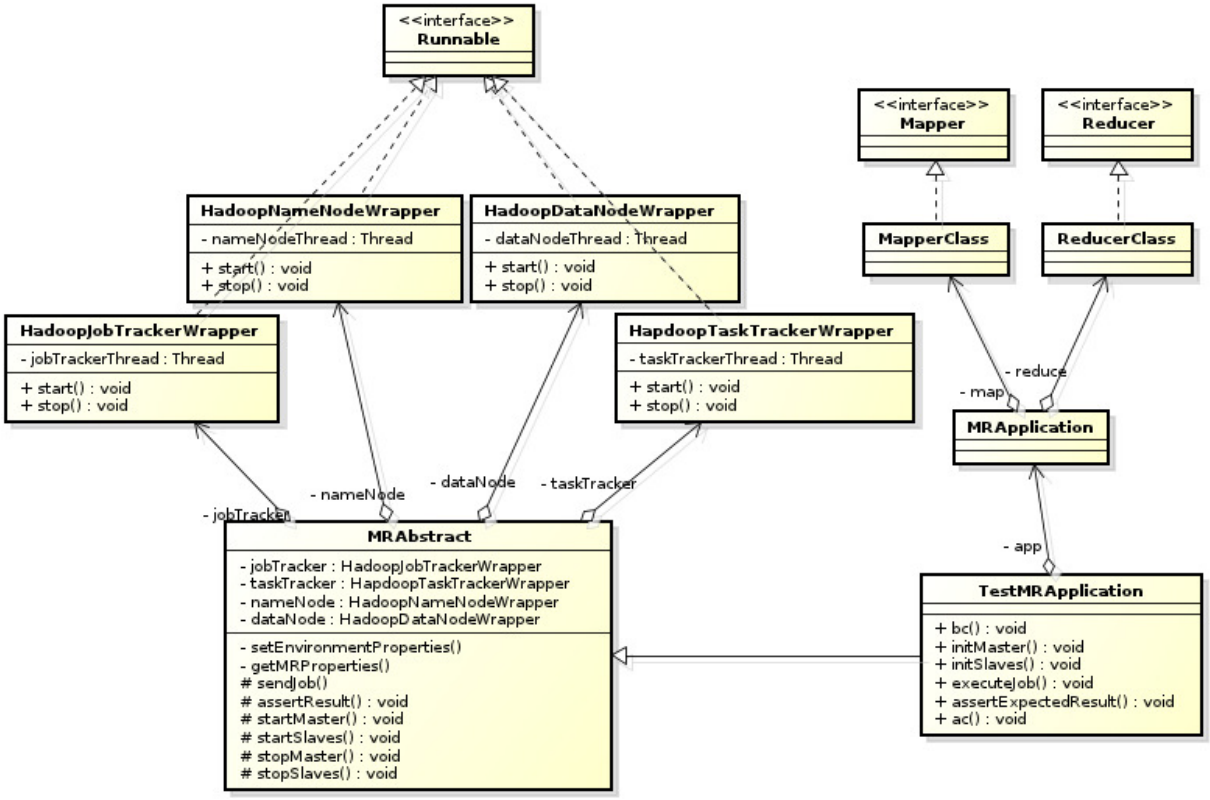


Figura 4.4: Diagrama de classes do HadoopTest.

A MRAbstract é a classe principal do HadoopTest, é através dela que todos os métodos para interações com o MapReduce são definidos. Na Figura 4.4 a classe MRAbstract faz uso de outras 4 classes: HadoopJobTrackerWrapper, HadoopTaskTrackerWrapper, HadoopNameNodeWrapper e HadoopDataNodeWrapper, que chamamos de classes *wrapper*. Estas classes fazem a interação do HadoopTest com a implementação MapReduce.

Neste trabalho as classes *wrapper* interagem somente com o Hadoop, não atendo-se a outras implementação de MapReduce. Nesta versão do HadoopTest, as interações com o MapReduce acontecem por meio das Interfaces de Programação (*Application Programmer Interface*, API) disponíveis no Hadoop. Para que o HadoopTest possa interagir com outras implementações do MapReduce basta que novas classes *wrappers* sejam criadas para elas. Estas novas classes irão interagir com a classe MRAbstract, sem que ela seja modificada, simplificando o processo de adaptação HadoopTest a outras implementações.

No HadoopTest, a classe *TestMRApplication* especifica o caso de teste. Ela estende a classe principal (*MRAbstract*) e utiliza a classe *MRApplication* como sistema MapReduce a ser testado. Portanto, a classe *MRApplication* é genérica a qualquer classe correspon-

dente a sistemas para MapReduce. Estes sistemas são divididos em funções Map e Reduce, que são implementadas através das classes *MapperClass* e *ReducerClass*, respectivamente.

4.3.2 Como Escrever Casos de Teste para o HadoopTest

Os casos de teste para o HadoopTest são criados através de uma classe Java. As ações do caso de teste correspondem aos métodos desta classe. Cada ação é identificada através de uma *anotação* Java predefinida, que precede o método. As anotações definem os parâmetros como o nome da ação, sua hierarquia e outros, anteriormente descritos na Definição 4.1.3. Baseando-se nas arquiteturas clássicas de controle de execução de testes [6], as anotações permitidas no HadoopTest são: *BeforeClass*, *Test* e *AfterClass*. A anotação *BeforeClass* é executada antes de todas as demais anotações e deixa o ambiente de execução (i.e., variáveis, carregamento de arquivos de configuração) pronto para que o HadoopTest seja executado. A anotação *Test* define as ações do caso de teste e a anotação *AfterClass*, executa depois de todas as outras anotações, limpando as alterações feitas pela anotação *BeforeClass* no ambiente de execução.

Para explicar a sintaxe utilizada na criação dos casos de teste para o HadoopTest será utilizado o caso de teste de exemplo mostrado na Tabela 4.1. Somente foi adicionado a ele as ações *bc* e *ac*, que correspondem às anotações *BeforeClass* e *AfterClass*, respectivamente.

A classe que implementa o caso de teste do HadoopTest herda os atributos e métodos da classe principal (AbstractMR), pois é ela quem fornece as interfaces para interação do HadoopTest com o MapReduce.

```
public class TestMRAppication extends AbstractMR {
    ...
}
```

O método **bc()**, que corresponde à anotação *@BeforeClass*, faz a leitura dos arquivos de configuração do HadoopTeste e deixa o ambiente pronto para que ele possa ser executado.

```
@BeforeClass(range="*", timeout=100000)
```

```
public void bc() {
  readConfiguration();
}
```

Em seguida, o componente principal do MapReduce é inicializado. Para isso, foi utilizado o método **inicializarM()**, sendo ele executado somente no testador t_0 (range="0") do HadoopTest, com hierarquia de nível 1 (order=1).

```
@TestStep(order=1, range="0", timeout=100)
public void inicializarM() {
   startMaster();
}
```

Se a ação *inicializarM* foi corretamente executada (depend="inicializarM"), então os componentes secundários do MapReduce são inicializados em todos os testadores do HadoopTest.

```
@TestStep(order=2, range = "*", timeout=1000, depend="a1")
public void inicializarW() {
   startSlaves ();
}
```

Caso a ação *inicializarW* tenha sido corretamente executada, então os componentes do MapReduce já estão executando corretamente. Assim, já é possível que as tarefas de processamento sejam enviadas a ele. Na ação **enviarTarefa()**, através do testador t_0, é enviado um trabalho para ser executado pelo MapReduce.

```
@TestStep(order=3, range="0", timeout=100000, depend="a2")
public void enviarTarefa() {
   sendJob();
}
```

Paralelamente à ação *enviarTarefo* o testador t_2 desconecta um dos componentes secundários do MapReduce, simulando uma falha de rede. O paralelismo destas duas ações é definido atribuindo a elas o mesmo nível hierárquico, como no caso, que ambas possuem o nível hierárquico 3 (order="3").

```
@TestStep(order=3, range="2", timeout=1000, depend="a2")
public void matarW() {
    dropWorker();
}
```

Quando as ações *enviarTarefa* e *matarW* terminam de executar, então o resultado da tarefa submetida pela ação *enviarTarefa* (i.e., output) é comparado com o resultado esperado (i.e., expected), para testar se a tarefa foi executada corretamente pelo MapReduce, mesmo com a falha de rede simulada através da ação *matarW*.

```
@TestStep(order=4, range="0", timeout=10000, depend="a3,a4")
public void verificarResultado() {
    assert(output, expected);
}
```

Após a comparação dos resultados ser feita, os componentes secundários do MapReduce são parados.

```
@TestStep(order=5, range="*", timeout=1000)
public void pararW() {
    stopSlaves();
}
```

Tendo os componentes secundários sido parados, então o mesmo é feito com o componente principal.

```
@TestStep(order=6, range="0", timeout=1000)
public void pararM() {
    stopMaster();
}
```

Por último, a ação *ac* desfaz as alterações do ambiente feitas pela ação *bc*, para que elas não interfiram na execução de outros casos de teste neste mesmo ambiente de execução.

```
@AfterClass(range="*", timeout = 100000)
public void ac() {
```

```
}
```

Seguindo essa estrutura, outros casos de teste podem ser criados para testar sistemas baseados em MapReduce. A sintaxe utilizada para criá-los é muito simples, basta herdar as propriedades da classe principal e definir as ações do caso de teste, através das anotações Java mostradas nesta sessão.

4.3.3 Contribuições

O HadoopTest é um controlador de testes para sistemas baseados em MapReduce, que possibilita tanto testar as implementações de MapReduce, como suas aplicações. Ele possibilita que sejam criados casos de teste para testes dinâmicos e estáticos de diversos tipos, abrangendo dos testes unitários aos testes de sistema, propostos pelo modelo V descrito em Amman e Offut [3]. Então, casos de teste com finalidades mais específicas (e.g., implementações MR, testar aplicações MR em diferentes níveis), podem ser escritos para serem executados no HadoopTest.

No HadoopTest, foi melhorada a sincronização de execução de ações de teste, proposta em Almeida [12]. Foi criado o conceito de dependência de ações, que evita a execução desnecessária de ações de teste após alguma ação ter recebido o veredito *erro*. Além disso, no modelo de sincronização de Almeida [12], uma mesma ação de teste pode ser executada paralelamente em diferentes testadores, porém isso não é possível quando se trata de duas ações de teste distintas. Esta característica também foi criada no HadoopTest, sendo possível através dele executar ações distintas de forma paralela.

Para testar sistemas baseados em MapReduce é importante controlar os componentes do sistema. No MapReduce, os componentes normalmente são executados em grande número, o que aumenta as chances deles falharem. Se os componentes não são controlados, então se torna difícil identificar quais são os componentes que falham, quando isso acontece. Outra situação abrangida no controle dos componentes é a simulação de algum comportamento do MapReduce. Determinados casos de teste são criados para verificar um sistema na presença de falhas (e.g., saída de algum componente da rede). Para que

falhas possam ser simuladas, os componentes do MapReduce precisam ser controlados. O HadoopTest controla cada um dos componentes do MapReduce, permitindo que eles recebam estímulos (e.g., simulação de comportamento) ou somente sejam monitorados.

CAPÍTULO 5

EXPERIMENTOS E VALIDAÇÃO

Neste capítulo são apresentados os experimentos utilizados para validar o HadoopTest. Inicialmente são apresentadas as duas aplicações utilizadas neles e na sequência, são descritos os experimentos realizados e seus resultados. Os experimentos foram executados sobre 3 computadores em uma rede local com barramento de 100 Mbs. Os computadores utilizavam o Sistema Operacional (SO) GNU/Linux Debian e cada um deles possuía a seguinte configuração: 1 processador *quad-core* Intel de 2.40 GHz, 2 GB de memória RAM e HDs SATA de 80 GB. Para implementar o ambiente MapReduce utilizamos o Hadoop versão 0.20.2.

5.1 Aplicações Utilizadas nos Experimentos

Nos experimentos foram utilizadas duas aplicações clássicas, distribuídas juntamente com o Hadoop, o *PiEstimator* e o *WordCount*. As duas aplicações são simples e através delas é possível realizar os experimentos para validar o HadoopTest.

O *PiEstimator* é utilizado para calcular o valor aproximado do π através do método estatístico de Monte Carlo. Este método baseia-se em um círculo inserido exatamente no meio de um quadrado de tamanho 1. A função Map do MapReduce cria pontos dentro de todo o quadrado, de forma randômica, e identifica quais destes pontos foram criados dentro do círculo e quais foram criados fora dele. Então, a função de Reduce acumula os pontos identificados como dentro do círculo (I) e os pontos fora dele (O) e realiza a fração I/T, onde $T = I + O$. O cálculo de I/T resulta na aproximação da área do círculo, sendo que ela corresponde a $\pi/4$. Então, para estimar o valor de π a função de Reduce multiplica a área aproximada do círculo por 4 ($4 * (I/T)$), gerando o resultado final, ou seja, o valor aproximado do π.

O *WordCount* é uma aplicação MapReduce que busca pelo total de ocorrências de

palavras distintas em um arquivo de texto. Nele, a função de Map recebe um pedaço do arquivo de texto de entrada e para cada palavra encontrada ele emite um par composto pela palavra e o número 1. Então, a função de Reduce conta a quantidade de cada palavra e emite o resultado final, que constitui-se de pares contendo cada palavra e o total de ocorrências dela no texto.

5.2 Validação da Não Intrusão do HadoopTest

Os primeiros experimentos objetivam identificar se o HadoopTest influencia no resultado dos testes e se ele compromete a escalabilidade dos sistemas MapReduce. Ambas as aplicações (PiEstimator e WordCount) foram executadas diretamente no Hadoop, sem o controle do HadoopTest, e posteriormente foram executadas pelo HadoopTest, por meio do caso de teste mostrado na Tabela 4.1 (sem a ação dropWorker). Nos experimentos foram considerados o tempo de execução e o resultado em ambas as execuções. Para o PiEstimator foram utilizadas 100 amostras (i.e., pontos no quadrado) para cada Map e variou-se o número de Maps de 50 a 2000. Os Maps são distribuídos entre os computadores que estão executando o MapReduce, em nosso caso, entre até 3 computadores, e conforme o número de Maps aumenta, também aumenta a carga entre estes computadores. Em uma execução com 2000 Maps tem-se uma carga considerável para verificar se o HadoopTest compromete a escalabilidade do MapReduce.

Primeiramente os testes foram executados em uma implementação MapReduce em um computador e depois, os mesmos testes foram executados em uma implementação utilizando 3 computadores. Todos os computadores utilizados possuem a mesma configuração, já descrita no início deste capítulo. Os resultados das execuções são mostrados nas Tabelas 5.2 (1 computador) e 5.2 (3 computadores), considerando o tempo de execução, o resultado e a quantidade de Maps de cada uma delas.

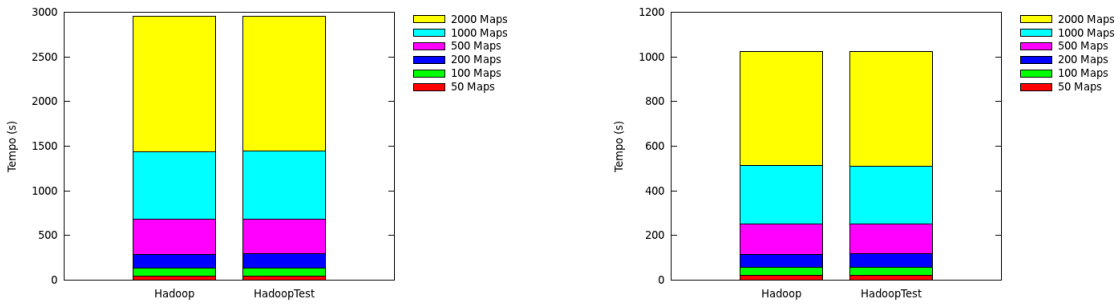
Tabela 5.1: Comparativo da execução do PiEstimator em 1 computador.

	Tempo de Execução	Pi calculado	Quantidade de Maps
Hadoop	92,873	3,1416	50
HadoopTest	93,966	3,1416	50
Hadoop	170,566	3,1408	100
HadoopTest	169,071	3,1408	100
Hadoop	320,039	3,1428	200
HadoopTest	318,391	3,1428	200
Hadoop	774,282	3,142480	500
HadoopTest	774,952	3,142480	500
Hadoop	1524,137	3,1412	1000
HadoopTest	1525,141	3,1412	1000
Hadoop	3034,72	3,14118	2000
HadoopTest	3032,059	3,14118	2000

Tabela 5.2: Comparativo da execução do PiEstimator em 3 computadores.

	Tempo de Execução	Pi calculado	Quantidade de Maps
Hadoop	44,785	3,1416	50
HadoopTest	45,505	3,1416	50
Hadoop	70,83	3,1408	100
HadoopTest	69,672	3,1408	100
Hadoop	118,15	3,1428	200
HadoopTest	119,36	3,1428	200
Hadoop	268,705	3,142480	500
HadoopTest	267,096	3,142480	500
Hadoop	522,957	3,1412	1000
HadoopTest	521,354	3,1412	1000
Hadoop	1021,33	3,14118	2000
HadoopTest	1022,645	3,14118	2000

Os tempos de execução das duas tabelas são ilustrados no gráfico da Figura 5.1. A Figura 5.1(a) compara os tempos de execução do Hadoop e do HadoopTest quando executados sobre uma implementação MapReduce com um computador e a Figura 5.1(b) compara os tempos de execução considerando uma execução em três computadores.



(a) 1 computador.

(b) 3 computadores.

Figura 5.1: Gráficos comparativos dos experimentos com o Hadoop e HadoopTest.

Através do comparativo da execução do PiEstimator verifica-se que os resultados das

execuções não foram influenciados e seus tempos de execução foram similares quando os experimentos foram executados com o HadoopTest. Os tempos de execução variam em alguns milesegundos do HadoopTest para o Hadoop, isso devido a fatores externos (e.g., rede de computadores, concorrência de outras aplicações no sistema operacional), porém esta variação é insignificante quando se trata de sistemas distribuídos. Na Figura 5.1 é possível perceber que a escalabilidade das execuções, com e sem o HadoopTest, também não foi comprometida, já que mesmo aumentando o número de maps, os tempos do Hadoop e do HadoopTest continuam similares. Para a aplicação WordCount também foram comparados o resultado e o tempo de execução e ambos também não sofreram influência do HadoopTest.

5.3 Validação do Controle do HadoopTest Sobre os Componentes do MapReduce

Os experimentos realizados nesta seção buscam validar o controle sobre os componentes do MapReduce, mesmo com a presença de falhas. Estes experimentos utilizaram o caso de teste de exemplo (Tabela 4.1) e incluem a ação *dropWorker*, que não foi utilizada anteriormente. No caso de teste exemplo, a ação dropWorker simula uma falha (i.e., interrompe a execução de uma dos componentes do MapReduce), então é testado se o resultado do sistema é alterado devido a esta falha. Segundo as especificações do MapReduce, descritas por Dean e Ghemawat [15], as falhas dos componentes devem ser tratadas pelo MapReduce e não devem influenciar no resultado dos sistemas.

Os experimentos descritos nesta seção também foram executados sobre uma implementação MapReduce com 3 computadores e o PiEstimator foi o sistema testado. Nos experimentos foram utilizados 1000 Maps e a cada um deles foram atribuídas 100 amostras (i.e., pontos no quadrado). A Tabela 5.3 mostra o resultado destes experimentos.

Tabela 5.3: Resultados do teste em um ambiente MapReduce com falhas.

	Tempo de Execução	Pi calculado	Quantidade de Maps
Sem falha	521,354	3,1412	1000
Com falha	843,182	3,1412	1000

Os resultados da tabela mostram que o PiEstimator teve seu tempo de execução aumentado, quando executado através do caso de teste que simula falhas, mas seu resultado não foi influenciado pela falha, confirmando as especificações descritas por Dean e Ghemawat [15]. O aumento no tempo de execução acontece devido aos mecanismos de tolerância a falhas do MapReduce, pois caso algum componente falha, então o MapReduce atribui as operações que ele estava executando a outro componente e a tarefa é processada novamente. Para que a simulação de falha seja possível é necessário que a ferramenta de teste controle o componente sobre o qual é simulada a falha. O HadoopTest conseguiu executar o caso de teste com simulação de falhas e validar o resultado do PiEstimator, comprovando sua habilidade em controlar os componentes do MapReduce.

5.4 Validação dos Resultados dos Casos de Teste

Todos os experimentos basearam-se no caso de teste de exemplo, mostrado na Tabela 4.1. Este caso de teste valida os resultados dos sistemas executados sobre o MapReduce, testando se eles são os mesmos que os resultados esperados. Porém, é preciso saber se a ferramenta de teste avalia os sistemas de forma correta e para isso é necessário que o HadoopTest seja validado.

Uma técnica utilizada para validação é o teste de mutações ou análise de mutações. Nos testes de mutação são injetados defeitos no sistema original, sendo que para cada defeito é originado um novo sistema, chamado de sistema mutado. Então, o sistema original é comparado com cada um dos sistemas mutados, avaliando se houve diferença entre eles. O resultado desta comparação define se uma ferramenta de teste é capaz de identificar o defeito no sistema mutado [24].

Para gerar as mutantes nos sistemas para MapReduce testados (i.e., PiEstimator e WordCount) foi utilizada a biblioteca Java ASM [8]. Os mutantes gerados nestes sistemas mudaram os operadores aritméticos (i.e., '+' por '-', 'verdardeiro' por 'falso') existentes neles. Para cada mutante foi criada uma nova classe Java, baseada na classe Java original do sistema.

Para avaliar se o HadoopTest é capaz de identificar os mutantes gerados, primeira-

mente foi executado o caso de teste com o sistema original e armazenou-se o resultado apresentado por ele. Depois, cada um dos sistemas mutados foi executado através do mesmo caso de teste e seu resultado foi armazenado. Quando todos os sistemas (i.e., original e mutados) foram executados, então comparou-se o resultado do sistema original com cada um dos sistemas mutados. Nas avaliações foram considerados três tipos de mutantes: **mutantes mortos**, **mutantes vivos** e **mutantes equivalentes**.

O mutante é dito como morto quando a ferramenta de teste é capaz de identificá-lo (i.e., distinguí-lo da aplicação original). O mutante vivo é aquele que não consegue ser identificado pela ferramenta de teste, porém pode ser identificado por outra ferramenta. Já o mutante equivalente é o que se refere aos operadores que possuem a mesma semântica dos operadores do sistema original (e.g., '>' e '>='). Estes mutantes não alteram o resultado do sistema e necessitam de casos de teste específicos para que possam ser identificados. Caso os mutantes equivalentes não sejam identificados, não trata-se de um problema da ferramenta de teste, mas sim, do caso de teste, que é pouco específico.

A Tabela 5.4 ilustra os resultados das execuções de 13 mutantes (M_0 a M_12) através do HadoopTest, sendo eles originários da aplicação PiEstimator.

Tabela 5.4: Resultado do caso de teste para 13 mutantes do PiEstimator.

Mutante	**Resultado**	**Identificação**
M0	Nulo	Identificado
M1	3.1416	Não identificado
M2	Nulo	Identificado
M3	Nulo	Identificado
M4	3.0776	Identificado
M5	3.1312	Identificado
M6	3.1416	Não identificado
M7	Nulo	Identificado
M8	3.1416	Não identificado
M9	3.1416	Não identificado
M10	3.1408	Identificado
M11	3.1408	Identificado
M12	3.1416	Não identificado

O caso de teste utilizado (Tabela 4.1) somente avalia o resultado das aplicações, então os mutantes que obtiveram o resultado igual ao do sistema original foram classificados como *identificados* e os demais, que tiveram resultados distintos ou nulos, consideramos

como não identificados. O valor aproximado do π retornado pela aplicação original foi *3.1416* e somente os mutantes M1, M6, M8, M9 e M12 retornaram este valor. Então, eles não conseguiram ser identificados pelo caso de teste, diferente de todos os outros mutantes. Os mutantes que não foram identificados são detalhados na Tabela 5.4.

Tabela 5.5: Descrição dos mutantes do PiEstimator

Mutante	**Operação**	**Mutação**
M1	$jobConf.setSpeculativeExecution(false)$;	Troca "false" por "true"
M6	$duration = (System.currentTimeMillis() - startTime)/1000.0$;	Troca "-" por "+"
M8	$fs.delete(TMP_DIR, true)$;	Troca "true" por "false"
M9	$fs.delete(TMP_DIR, true)$;	Troca status de $fs.delete$ de 1 para 0
M12	$return0$;	Retorna 1

Através da Tabela 5.4 pode-se observar que todos os mutantes classificados como não identificados podem ser considerados mutantes equivalentes. Isso porque todos eles possuem a mesma semântica que a parte de código correspondente a eles, no sistema original. O mutante M1 somente troca falso por verdadeiro no método que faz com que múltiplas tarefas executem a mesma atividade, utilizado para implementar tolerância a falhas; O mutante M6 altera o operador aritmético "-" pelo operador "+", no cálculo do tempo de execução; O mutante M8 troca verdadeiro por falso no método que remove diretórios temporários; O mutante M9 troca o status da operação de exclusão do diretório temporário; E o mutante M12 somente muda o retorno no final do método que executa o cálculo do π. Portanto, nenhum destes mutantes altera significativamente o código a ponto de mudar o resultado do cálculo do π. Dessa forma, como nosso caso de teste somente avalia o resultado final, eles não puderam ser identificados. Para que eles pudessem ser identificados seriam necessários casos de teste mais específicos.

Para o WordCount foram gerados oito mutantes, entre eles cinco foram detectados pelo HadoopTest. Todos os mutantes não detectados pelo HadoopTest também foram considerados como equivalentes, pois geraram mutações com a mesma semântica que a aplicação original.

A Figura 5.2 ilustra em forma de gráfico os resultados obtidos com o HadoopTest na

análise de mutantes das aplicações PiEstimator e WordCount.

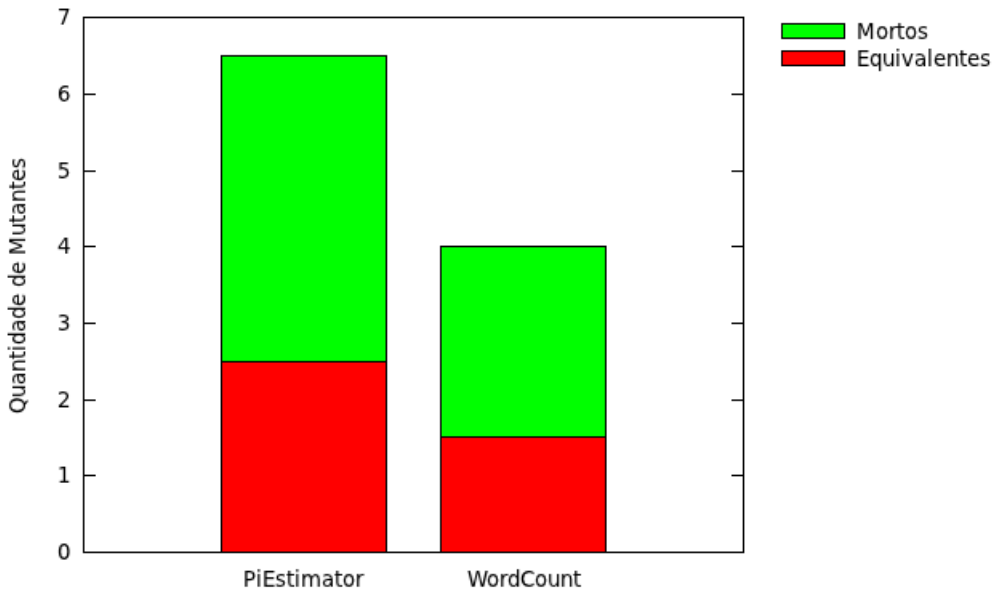


Figura 5.2: Quantidade de mutantes mortos e equivalentes.

O gráfico demonstra que a maioria dos mutantes foram mortos pelo HadoopTest. Todos os mutantes não identificados foram considerados como equivalentes e não foram identificados devido a abrangência restrita de nosso caso de teste. Para que eles possam ser identificados é necessário um caso de teste mais específico, que avalie a sintaxe das aplicações para MapReduce, por exemplo. Sendo assim, se forem desconsiderados os mutantes equivalentes, o Hadooptest foi capaz de identificar todos os demais defeitos das aplicações mutadas.

CAPÍTULO 6

CONCLUSÃO

O MapReduce é um *framework* para processamento de grandes quantidades de dados, que tem se tornado popular nos últimos anos. Ele normalmente é executado sobre um agrupamento de centenas ou milhares de computadores, o que aumenta a possibilidade de falhas [15] (i.e., problemas de hardware, problemas de rede), que podem comprometer a execução e o resultado dos sistemas baseados em MapReduce (i.e., sistemas criados para executarem sobre o MapReduce). Para avaliar se as falhas comprometem a confiabilidade destes sistemas, eles precisam ser testados. Porém, as soluções para testes de sistemas baseados em MapReduce são poucas e não o fazem de uma maneira eficiente.

Neste trabalho foi apresentado o HadoopTest, um controlador de testes para sistemas baseados em MapReduce. Ele combina a simulação de falhas com testes funcionais, possibilitando a criação de casos de teste complexos. Os casos de teste para HadoopTest podem testar tanto as implementações do MapReduce, como as aplicações que são executadas sobre elas. O HadoopTest coordena a execução de casos de teste sobre os componentes do MapReduce, que organizam-se de forma distribuída. Isso é feito controlando cada componente do MapReduce através de testadores distribuídos, onde cada testador inicializa um componente do MR, o monitora e é capaz de simular falhas nele. Além disso, ele melhora a sincronização de execução de ações de teste proposta por Almeida [12], criando a dependência de ações e a execução paralela de ações de teste.

Através dos experimentos executados, foi mostrado como o HadoopTest coordena casos de teste distribuídos sobre os componentes do MapReduce. Também foram feitos testes para verificar se o MapReduce realmente é tolerante a falhas, através de um caso de teste que remove um dos componentes do MR enquanto um sistema é executado. O HadoopTest também foi utilizado para testar o PiEstimator e o WordCount, duas aplicações distribuídas com o Hadoop, uma implementação MapReduce de código aberto, mantida pela

fundação Apache. Para validar os resultados conseguidos com o HadoopTest foram criadas mutações nas aplicações testadas. Através da análise de mutações o HadoopTest demostrou que está apto a identificar falhas, encontrando todas os mutantes não equivalentes.

Como trabalho futuro, planejamos criar uma interface para injeção de falhas para o HadoopTest. Os experimentos realizados somente simularam uma falha de rede (i.e., desconexão do componente do MapReduce) através de uma ação de teste e isso foi feito sem uma interface específica para injeção de falhas. Uma interface para injeção de falhas melhora o controle das falhas inseridas no SUT, pois pode definir, por exemplo, quais são as falhas possíveis de serem injetadas sobre o SUT, os erros que elas ocasionam e os componentes sobre os quais elas devem ser injetadas . Também deve ser criada uma estrutura para o monitoramento do *workflow* do SUT (e.g., quais são os Maps executando, quem está se comunicando, quem é o próximo a processar). Dessa forma, casos de teste mais complexos podem ser criados, baseando-se no estado de execução do *workflow*. Além disso, devem ser testadas aplicações MapReduce mais complexas, como o Hive, o Mahout e o Nutch.

BIBLIOGRAFIA

[1] P2: Declarative networking. http://p2.cs.berkeley.edu/, 2011.

[2] ISO International Standard 9646. Conformance testing methodology and framework, 1991.

[3] Paul Ammann e Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[4] Apache. Hadoop website. http://hadoop.apache.org, 2010.

[5] Apache. Herriot, large-scale automated test framework. http://wiki.apache.org/hadoop/, 2010.

[6] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] Francisco Brasileiro, Patricia Machado, Walfredo Cirne, e Alexandre Duarte. Gridunit: software testing on the grid. *Software Engineering, International Conference on*, 0, 2006.

[8] Eric Bruneton, Romain Lenglet, e Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. *In Adaptable and extensible component systems*, 2002.

[9] Bogdan Butnaru, Florin Dragan, Georges Gardarin, Ioana Manolescu, Benjamin Nguyen, Radu Pop, Nicoleta Preda, e Laurent Yeh. P2ptester: a tool for measuring p2p platform performance. *ICDE*. IEEE, 2007.

[10] Cloudera. Mrunit project. http://archive.cloudera.com/docs/mrunit/index.html, 2010.

[11] Aster Data. Aster mapreduce analytics portifolio. http://www.asterdata.com/product/advanced-analytics.php, 2011.

[12] Eduardo Cunha de Almeida. *Testing and Validation of Peer-to-Peer Systems*. Tese de Doutorado, Université de Nantes, 2009.

[13] Eduardo Cunha de Almeida, João Eugenio Marynowski, Gerson Sunyé, e Patrick Valduriez. Peerunit: a framework for testing peer-to-peer systems. Charles Pecheur, Jamie Andrews, e Elisabetta Di Nitto, editors, *ASE*. ACM, 2010.

[14] Jeffrey Dean e Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, 2004.

[15] Jeffrey Dean e Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51, January de 2008.

[16] Márcio Eduardo Delamaro, José Carlos Maldonado, e Mario Jino. *Introdução ao Teste de Software*. Elsevier, 2007.

[17] Erlang. Erlang programming language. http://www.erlang.org, 2011.

[18] Sanjay Ghemawat, Howard Gobioff, e Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5), 2003.

[19] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, e Colin Willcock. An introduction to the testing and test control notation (ttcn-3). *Comput. Netw.*, 42, June de 2003.

[20] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, e Koushik Sen. Towards automatically checking thousands of failures with micro-specifications, Jun de 2010.

[21] Daniel Hughes, Phil Greenwood, e Geoff Coulson. A framework for testing distributed systems. Germano Caronni, Nathalie Weiler, e Nahid Shahmehri, editors, *Peer-to-Peer Computing*. IEEE Computer Society, 2004.

[22] Gregory M. Kapfhammer. Automatically and transparently distributing the execution of regression test suites. *Proceedings of the 18th International Conference on Testing Computer Software*, Washington, D.C., June de 2001.

[23] Yunhao Liu, Xiaomei Liu, Li Xiao, Lionel M. Ni, e Xiaodong Zhang. Location-aware topology matching in p2p systems. *INFOCOM*, 2004.

[24] Yu-Seung Ma, Jeff Offutt, e Yong-Rae Kwon. Mujava: a mutation system for java. *Proceedings of the 28th international conference on Software engineering*, ICSE '06, New York, NY, USA, 2006. ACM.

[25] Oracle. Jpda. http://download-llnw.oracle.com/javase/1.4.2/docs/guide/jpda/, 2010.

[26] Xinghao Pan, Jiaqi Tan, Soila Kavulya, Rajeev G, e Priya Narasimhan. Ganesha: Black-box fault diagnosis for mapreduce systems, 2008.

[27] Xinghao Pan, Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, e Priya Narasimhan. Ganesha: blackbox diagnosis of mapreduce systems. *SIGMETRICS Perform. Eval. Rev.*, 37(3), 2009.

[28] Tauhida Parveen, Scott Tilley, Nigel Daley, e Pedro Morales. Towards a distributed execution framework for junit test cases. *ICSM*. IEEE, 2009.

[29] Roger S. Pressman. *Engenharia de Software*. Pearson Makron Books, 1995.

[30] Thomas Rings, Helmut Neukirchen, e Jens Grabowski. Testing grid application workflows using ttcn-3. *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, Washington, DC, USA, 2008. IEEE Computer Society.

[31] Atul Singh, Petros Maniatis, Timothy Roscoe, e Peter Druschel. Using queries for distributed monitoring and forensics. Yolande Berbers e Willy Zwaenepoel, editors, *EuroSys*. ACM, 2006.

[32] Sun. Junit website. http://www.junit.org, 2010.

[33] The Apache Software Foundation. Apache hadoop. http://hadoop.apache.org/, 2011.

[34] The Apache Software Foundation. Apache hadoop users. http://wiki.apache.org/hadoop/PoweredBy/, 2011.

[35] Andreas Ulrich e Hartmut König. Architectures for testing distributed systems. Gyula Csopaki, Sarolta Dibuz, e Katalin Tarnay, editors, *IWTCS*, volume 147 of *IFIP Conference Proceedings*. Kluwer, 1999.

[36] Jason Venner. *Pro Hadoop - Pro Hadoop - Build Scalable, Distributed Applications in the Cloud*. Apress, 2009.

[37] Thomas Walter, Ina Schieferdecker, e Jens Grabowski. Test architectures for distributed systems: State of the art and beyond. *IWTCS*, 1998.

[38] Richard Winter e Pekka Kostamaa. Large scale data warehousing: Trends and observations. Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, e Vassilis J. Tsotras, editors, *ICDE*. IEEE, 2010.

[39] Ji Wu, Liu Yang, e Xu Luo. Jata: A language for distributed component testing. *Asia-Pacific Software Engineering Conference*, 0, 2008.

[40] Zhizhi Zhou, Hao Wang, Jin Zhou, Li Tang, Kai Lim, Weibo Zheng, e Meiqi Fang. Pigeon: a framework for testing peer-to-peer massively multiplayer online games over heterogeneous network. *Dans Consumer Communications and Networking Conference*, 2006.