Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Uma Abordagem para o Desenvolvimento de Testes de Sistema Automáticos de Aplicações Distribuídas

Giovanni Farias da Silva

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Dalton Serey e Raquel Lopes
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S586a Silva, Giovanni Farias da.

Uma abordagem para o desenvolvimento de testes de sistemas automáticos de aplicações distribuídas / Giovanni Farias da Silva. – Campina Grande, 2011.

120 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Orientadores: Prof. Dr. Dalton Serey Guerrero, Profª. Drª. Raquel Vigolvino Lopes.

Referências.

1. Redes de Computadores. 2. Sistemas Distribuídos. I. Título.

CDU 004.7 (043)

"UMA ABORDAGEM PARA O DESENVOLVIMENTO DE TESTES DE SISTEMA AUTOMÁTICOS PARA APLICAÇÕES DISTRIBUÍDAS"

GIOVANNI FARIAS DA SILVA

DISSERTAÇÃO APROVADA EM 31.08.2011

DALTON DARIO SEREY GUERRERO, D.Sc
Orientador(a)

RAQUEL VIGOLVINO LOPES, D.Sc
Orientador(a)

LÍVIA MARIA RODRIGUES SAMPAIO CAMPOS , D.Sc
Examinador(a)

AYLA DÉBORA DANTAS DE SOUZA REBOUÇAS, D.Sc
Examinador(a)

CAMPINA GRANDE - PB

Resumo

A atividade de testes vem sendo cada vez mais utilizada para assegurar que o produto atende às especificações do cliente. Teste de sistema de software é aquele destinado a testar a aplicação completa e integrada, como também deve ser executado em condições similares às quais o produto será submetido quando em produção. A automatização dos testes é um recurso amplamente desejado por reduzir drasticamento o custo envolvido na execução dos mesmos. Contudo, produzir testes automáticos é, muitas vezes, inviável devido a falta de ferramentas que deem ao desenvolvedor a possibilidade de configurar, implantar e executar o software a ser testado da forma desejada. Considerando o contexto de aplicações distribuídas, a realização de testes, manuais ou automáticos, dificulta significativamente devido às suas características de concorrência e distribuição. Essa dissertação apresenta uma abordagem de desenvolvimento de testes de sistema automáticos para aplicações distribuídas. O objetivo dessa abordagem é permitir que o programador escreva testes de sistema automáticos para sua aplicação utilizando a mesma linguagem de programação e ambiente de desenvolvimento utilizados durante a implementação da própria aplicação a ser testada. Para dar suporte à abordagem apresentada, foi desenvolvido um aplicativo intitulado SysTest. Uma avaliação levando em consideração a usabilidade do SysTest foi efetuada com o objetivo de mostrar a viabilidade da abordagem para o desenvolvimento de testes de sistemas reais e em produção. Os resultados obtidos dão indícios de que a API do SysTest permite a escrita de testes automáticos de aplicações distribuídas e é fácil de ser manipulada pelos programadores.

Abstract

The testing activity is being increasingly used to ensure that the software product meets customer specifications. System tests are those that aim at testing the whole application, fully integrated and running on an environment very similar to the one the application will be subjected when in production. Automating such tests is a very desirable requirement because automation can drastically reduce the testing execution costs. However, producing automated testing is often infeasible due to lack of tools that give developers the ability to configure, deploy and run the software under test as desired. By considering the context of distributed applications, the testing activity, whether manual or automatic, becomes considerably difficult due to distributed applications features such as concurrency and parallelism. This thesis presents an approach for developing automated system tests for distributed applications. The aim of this approach is to allow the programmer to write automated system tests for distributed applications using the same programming language and environment used to implement the application under test. In order to study the proposed approach we developed a prototype called SysTest. We carried out a usability evaluation of SysTest aiming at showing feasibility of our testing approach to the development of system tests for real distributed applications. The results provide evidence that the SysTest API allows the writing of automated tests for distributed applications and is easy to be used by the programmers that have developed the system under test.

Agradecimentos

Gostaria de agradecer primeiramente a Deus, por te me dado a oportunidade de realizar esse mestrado e de poder aprender tantas coisas. Também não teria conseguido essa conquista sem a contribuição de pessoas especiais. Abaixo, meus agradecimentos:

- À minha família (pais, irmãos, avós, tios, tias, primos e primas) por ter me dado condições para que eu terminasse o mestrado, além de sempre acreditar em mim e me incentivar. Em especial a minha mãe e ao meu pai, que já não está entre nós, que nunca mediram esforços para o estudo de seus filhos. Dedico essa conquista a vocês;
- À Thaís, minha riqueza, por ter ficado ao meu lado e compreendido minhas ausências;
- Aos meus orientadores, Dalton e Raquel, pelas suas recomendações e discussões ao longo do trabalho;
- Aos participantes do experimento, sem estes a avaliação do trabalho não poderia ter sido realizada;
- À todos os membros do LSD, inclusive aos que já sairam, pelas contribuições na minha pesquisa e pela amizade que construímos. Em especial àqueles aos quais convivi mais de perto: Edigley, Ciçu, Tomás, Fábio, Ana Cristina, Lauro, Leonardo, Paulo Ditarso, João Arthur, Manel, Carlinha, Pryscilla, Nazareno, Jhonny, Abmar, Ricardo, Mila, Guga, Fireman, Zé Flávio, Ayla, Fubica, Eliane, Marcelo Iuri, Antônio, Jaindson, Lesandro e David.
- Aos meus amigos do prédio, João Carlos, João Neto, Josué, Dudinha, Lulinha, Bárbara, Léo, Chinei, Danilo, Marcelinho, Mariela, Julian, Hélder, Rebeca, Heloiza, Lucyjane e Naia, pelos momentos de descontração e confraternização que vivemos ao longo desse período;

- Aos amigos que fiz em Campina Grande, PDT, Vinas, Charlesbel, Ciçu, Jemão, Gabriel, Paulo Rômulo, Manel Meota, André, Eduardo, Diego, Tássio, Italo, Gustavo, Kiko, Saulo e Rafael, pelas conversas, cervejadas, favadas e feijoadas que fizemos durante o período que estive em Campina Grande;

- À todos que fazem as Escolas Técnica Estadual (ETE) de Limoeiro-PE e Surubim-PE, em especial a Karina, Lúcia (minha mãe), Milton, Laisa, Keila e Manuela, por sua compreensão em meus momentos de ausência devido à dedicação a este trabalho de dissertação;

- À CAPES, pelo apoio financeiro durante parte desse trabalho.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Capítulo 1

Introdução

1.1 Contextualização

Para desenvolver software de qualidade é preciso garantir que os requisitos do software serão satisfeitos. Nesse contexto, a realização de testes vem sendo cada vez mais utilizada para garantir que o produto atenda às especificações do cliente [9]. Essa atividade consiste em definir casos de testes, exercitar o software como especificado e examinar se os resultados produzidos são os esperados [27].

Testes de sistema são aqueles projetados para testar a aplicação completa e integrada, avaliando se a mesma atende aos seus requisitos como um todo [24]. Esssa categoria de teste é muito importante, de modo que o sistema deve ser verificado sob condições similares aquelas a que ele será submetido quando estiver em produção. A automatização dos testes de sistema reduz bastante o custo envolvido na execução dos mesmos [41]. Por isso, a escrita de testes automáticos é uma atividade crítica, que merece atenção especial e suporte adequado [41].

Em se tratando de aplicações distribuídas, dois aspectos tornam as fases de desenvolvimento e teste mais complexas: concorrência e distribuição. A concorrência é caracterizada por diferentes linhas de execução que disputam pelo uso do mesmo recurso. Já distribuição é a característica de diferentes componentes do software estarem executando em máquinas diferentes [25]. Devido a essas particularidades, controlar totalmente a implantação e execução desse tipo de sistema é difícil. Então, a realização de atividades de testes de aplicações distribuídas torna-se uma atividade complexa por conta da existência de diversos compo-

nentes "espalhados"que devem cooperar entre si e da dificuldade de controlar totalmente a execução dessa aplicação.

Escrever testes automáticos é muitas vezes inviável, devido à falta de ferramentas que dêem ao programador a possibilidade de configurar, implantar e executar o software da forma desejada, bem como verificar se o seu comportamento foi de acordo com o esperado. No caso de aplicações distribuídas, as características supracitadas dificultam significativamente a atividade de teste desse tipo de software e, em especial, a realização de testes automáticos.

É importante ressaltar que a realização de testes de aplicações distribuídas, especialmente testes de sistema, é uma atividade de extrema importância para garantir a qualidade do software. Durante a fase de testes de sistema é possível identificar erros que só ocorrem quando a aplicação executa em condições similares àquelas a que será submetida quando estiver em produção. Porém, no caso de aplicações distribuídas, as características dificultam a realização dessa atividade.

Considerando o cenário de realização de testes automáticos de sistema para aplicações distribuídas, foi realizado um *survey* com alguns desenvolvedores de aplicações distribuídas com o objetivo de entender como os testes de sistemas de suas aplicações vem sendo realizados, e, quais as ferramentas mais conhecidas que os auxilia durante essa atividade.

1.2 Caracterização do Problema

Os resultados levantados dão indícios de que grande parte dos projetos envolvendo aplicações distribuídas realizam testes de sistema. No entanto, a grande maioria dos que fazem, executam manualmente ou de forma parcialmente automática com o uso de *shell scripts* [1].

O uso de *scripts* para automatizar testes de sistema implica em usar uma outra linguagem de programação para a implementação do teste, resultando na mudança de ambiente de desenvolvimento. Por essa razão, supõe-se que o uso desse recurso, além de não garantir a total automação do teste de sistema, traz uma série de dificuldades, tanto de implementação e manutenção dos casos de testes durante a evolução do software quanto de legibilidade/clareza dos testes por parte dos integrantes da equipe, que podem não estar acostumados com

[1] Não conseguem automatizar a execução de 100% do caso de teste, mas para o trecho automático usa *shell script*

esse outro ambiente de desenvolvimento.

O estudo realizado também apontou que grande parte dos programadores não conhece qualquer ferramenta disponível na comunidade que auxilie o desenvolvimento de testes automáticos de aplicações distribuídas. Os que afirmam conhecer, mencionaram ferramentas que podem apoiar em alguns aspectos, considerando as funcinalidades de cada uma, o programador que quer automatizar os casos de testes de seu software. Entretanto, o propósito principal das mesmas não é a automatização de testes. A metodologia desse estudo, bem como o detalhamento dos resultados obtidos serão apresentados no Capítulo 2.

É possível caracterizar e classificar as dificuldades relacionadas à atividade de testes de aplicações distribuídas em duas categorias distintas:

1. O controle e manipulação da infraestrutura onde o sistema que será testado será implantado para a sua execução;

2. A manipulação e consulta sobre o comportamento do próprio sistema sob teste (do inglês, SUT - *System Under Test*).

Este trabalho direcionou-se ao segundo problema. Ou seja, considerando que a infraestrutura esteja montada e configurada para a execução do teste, a questão está em como possibilitar que o testador desenvolva e execute testes de sistema de maneira automática a partir de sua máquina e usando o mesmo ambiente de desenvolvimento que foi utilizado para o desenvolvimento do SUT. Para que seja possível que a execução do teste seja automática, é necessário que o ambiente de execução também disponibilize uma maneira que permita o testador implantar o SUT nesse ambiente programaticamente. Portanto, mesmo fazendo parte do primeiro problema, este trabalho também precisou disponibilizar um meio para que o testador implantasse sua aplicação.

1.3 Solução Proposta

Considerando o escopo do problema escolhido para esse trabalho, apresentamos uma abordagem para desenvolvimento de testes de sistema automáticos de aplicações distribuídas. De acordo com nossa técnica, o testador deve poder escrever testes de sistema automáticos para o SUT utilizando a mesma linguagem de programação e ambiente de desenvolvimento

usados para a implementação do SUT. A idéia é que existam agentes de teste em execução nas máquinas que serão utilizadas durante o teste, e que o programador envie comandos para esses agentes de forma que ele possa exercitar o SUT. O objetivo dessa técnica é que o testador consiga manipular e consultar o SUT "espalhado" pela infraestrutura de teste a partir de sua máquina e utilizando uma linguagem de programação e ambiente de desenvolvimento já conhecidos.

Como prova de conceito, foi desenvolvido um protótipo, intitulado SysTest, que dá suporte a escrita de testes automáticos para aplicações distribuídas desenvolvidas em Java. As abstrações e métodos disponibilizados por este aplicativo foram baseados nas idéias e requisitos presentes na abordagem proposta.

1.4 Resultados

Dois estudos foram realizados para avaliar a abordagem proposta. Esses estudos foram conduzidos para avaliar: a viabilidade de escrever diferentes testes de sistema automáticos de uma aplicação distribuída através do SysTest; e, a aplicabilidade da abordagem no que diz respeito ao desenvolvimento de testes automáticos por parte de equipes de desenvolvimento de softwares reais e em produção.

Para avaliar a viabilidade da escrita de testes automáticos seguindo a abordagem proposta por meio do SysTest, inicialmente desenvolvemos uma aplicação distribuída fictícia. Em seguida definimos e implementamos casos de teste para a mesma usando as abstrações e métodos disponibilizados pelo SysTest, como também o *framework* JUnit [18] para checagem e publicação dos resultados. Esse estudo serviu para mostrar que é possível desenvolver diferentes testes automáticos para softwares distribuídos usando o SysTest e o mesmo ambiente de desenvolvimento do SUT.

Com relação à aplicabilidade da abordagem de testes, foi conduzido um experimento em que 8 programadores de 2 projetos de aplicações distribuídas foram orientados a escrever um caso de teste de sistema para sua aplicação utilizando a API (do inglês, *Application Programming Interface*) do SysTest. O experimento consistiu em utilizar o protocolo *Think Aloud* [14] para obtenção de dados a respeito das expectativas, estratégias, dificuldades e objetivos dos desenvolvedores ao utilizarem a API do SysTest para a implementação dos

testes. Os resultados desse experimento apontam que os desenvolvedores não tiveram dificuldades em utilizar a API e todos, sem exceção, a consideraram fácil de ser manipulada. Os resultados dos dois estudos realizados serão mais detalhados no Capítulo 5.

1.5 Estrutura do documento

Essa dissertação está organizada da seguinte maneira:

Capítulo 2. Nesse capítulo são apresentados os principais conceitos relativos a testes, sistemas distribuídos e avaliação de API. Além disso, também é apresentado um estudo introdutório realizado com o intuito de entender como as aplicações distribuídas estão sendo testadas.

Capítulo 3. Nesse capítulo a abordagem de teste é apresentada com suas características e requisitos.

Capítulo 4. Nesse capítulo será apresentado o estudo de caso implementado em Java, intitulado de SysTest, para a demonstração de viabilidade da abordagem.

Capítulo 5. Nesse capítulo apresentamos a metodologia utilizada para avaliar o trabalho, bem como os resultados oriundos dessa avaliação.

Capítulo 6. Nesse capítulo são apresentadas as conclusões e os trabalhos futuros.

Capítulo 2

Fundamentação Teórica e Estudo Introdutório

Nesse capítulo são apresentados os principais conceitos necessários ao entendimento dessa dissertação, bem como um estudo motivacional realizado no início do trabalho. Inicialmente discute-se sobre testes de software, testes de sistema, aplicações distribuídas, automatização de testes e avaliações de API. Em seguida, é apresentado o estudo inicial realizado para entender como as aplicações distribuídas são testadas em projetos reais.

2.1 Testes de Software

Para desenvolver software de qualidade é preciso garantir que os requisitos do software serão satisfeitos. Nesse contexto, a atividade de testes vem sendo cada vez mais importante para garantir que o produto atenda às necessidades do cliente [9].

Testar um software significa verificar através de uma execução controlada se o seu comportamento corresponde ao especificado. Essa tarefa consiste em projetar casos de testes, exercitar o software e examinar se os resultados produzidos estão de acordo com o especificado [27]. Portanto, a implementação de testes é uma atividade do processo de desenvolvimento de software extremamente importante para determinar a existência de erro no programa [23], e tem como principal objetivo encontrar o número máximo de erros dispondo de esforço mínimo.

Os casos de testes podem ser projetados para verificar diferentes porções do programa,

como também requisitos funcionais e/ou não-funcionais específicos. Logo, os testes de software podem ser classificados de acordo com seu objetivo particular. Dentre as categorias mais comuns, tem-se:

- Testes de Unidade
- Testes de Integração
- Testes de Sistema
- Testes de Aceitação

Este trabalho direciona-se aos testes de sistemas, que estão apresentados na subseção a seguir.

2.1.1 Testes de Sistemas

Teste de sistema de um software é aquele destinado a testar a aplicação completa e integrada, avaliando se a mesma atende aos seus requisitos como um todo [24]. Esse tipo de teste deve ser executado em condições, geralmente ambiente, similares àquelas em que o produto será submetido em um ambiente real, isto é, o ambiente onde o mesmo será executado quando estiver em produção. Por isso, testes de sistema são fundamentais na produção de software, pois podem identificar defeitos que só ocorrem quando a aplicação é exercitada por inteiro em seu ambiente de execução.

2.2 Aplicações Distribuídas

Considerando aplicações distribuídas, verificam-se tipicamente dois aspectos: concorrência e distribuição. Por concorrência entende-se o fato de que diferentes linhas de execução do sistema disputam o uso do mesmo recurso; e distribuição observa-se quando os diferentes componentes do software estão executando em máquinas diferentes [25]. Além disso, as aplicações distribuídas são inerentemente não-determinísticas, fato que aumenta a probabilidade de existência de falsos positivos durante um teste. Um falso positivo ocorre quando acontece uma falha no teste sem que a falha seja defeito do sistema, mas sim por problemas no teste (por exemplo, a verificação foi realizada no momento inadequado, pois a operação

anterior ainda não tinha concluído a execução). Falsos positivos são comuns no caso de sistemas que envolvem operações assíncronas.

Por conta dessas características, é difícil controlar totalmente a execução desse tipo de sistema. Consequentemente, a realização de atividades de testes de aplicações distribuídas torna-se uma atividade não trivial devido à existência de diversos componentes "espalhados" que devem operar em conjunto, como também, à falta de controle total da mesma.

2.3 Automatização de Testes

A automatização de testes é um recurso amplamente desejado por reduzir drasticamente o custo envolvido na execução dos mesmos [41]. Testes automáticos são escritos na forma de programas e podem ser executados e re-executados inúmeras vezes a um custo muito baixo, restringindo-se essencialmente, ao custo de ciclos da CPU que, em geral, é muito inferior ao de horas de um testador [8]. Portanto, a escrita de testes automáticos é uma atividade crítica que merece atenção especial e suporte adequado [41] [9].

Os testadores, tanto de aplicações locais como também distribuídas, podem fazer uso da automatização de testes para agilizar a execução dos mesmos. Contudo, produzir testes automáticos é, em alguns casos, inviável, devido à falta de ferramentas que dêem ao desenvolvedor a possibilidade de configurar, implantar e executar o software a ser testado da forma desejada, bem como verificar se o comportamento foi de acordo com o esperado. Além disso, as características de concorrência e distribuição intrínsecas dos sistemas distribuídos dificultam significativamente o teste desse tipo de software e, em especial, a escrita de testes automáticos.

2.4 Testes de Sistemas de Aplicações Distribuídas

A realização de testes de aplicações distribuídas é uma tarefa não trivial, pois as carcterísticas de distribuição e concorrência a dificultam. Além dessas particularidades, a probabilidade da existência de falsos-positivos faz com que a atividade de testes de aplicações distribuídas seja suscetível a erros. Existem alguns trabalhos relacionados com essa área de pesquisa. Dantas et al [1] buscam evitar que asserções em testes de sistemas multi-threaded sejam

feitas cedo ou tarde demais através de uma abordagem baseada na monitoração e controle das threads da aplicação. Outra opção para os programdores para tentar evitar o problema de falsos positivos é os desenvolvedores explicitamente pararem a execução do teste por um determinado tempo, que pode ou não ser suficiente para a finalização das execuções das operações anteriores.

Apesar das dificuldades para realizar testes de sistema para aplicações distribuídas, essa é também uma atividade de extrema importância para garantir a qualidade do software, pois pode identificar defeitos que só ocorrem quando a aplicação é exercitada por inteiro em seu ambiente de execução. Em vista disso, decidiu-se realizar um *survey* entre a comunidade que desenvolve software distribuído com a finalidade de verificar se: (i) testes de sistemas são realizados; (ii) como são executados (automaticamente ou manualmente); (iii) e, o conhecimento de ferramentas que possibilitem a automatização de testes para aplicações dessa natureza por parte da comunidade.

A metodologia de realização desse estudo introdutório bem como os resultados obtidos serão apresentados e analisados em 2.6.

2.5 Avaliação de Usabilidade de APIs

O presente trabalho apresenta uma abordagem de teste que utiliza uma API para permitir a escrita de testes de sistema para aplicações distribuídas que possam ser executados de forma automática. A abordagem proposta nesta dissertação apoia-se nesta API e por esta razão é necessária uma avaliação que explore aspectos de usabilidade desta API.

Uma metodologia de avaliação de APIs foi adotada por estudos realizados pela Microsoft [14] [15] [13], e, posteriormente reproduzidos por outros pesquisadores [43] [30] [45] [44] [5] [28] de reconhecida competência nessa área. Todos esses trabalhos avaliam a usabilidade de uma API considerando a ideia de que a comparação entre o que os programadores esperam e o que a API provê é a questão interessante enquanto avaliando a usabilidade de uma API [14].

O protocolo *Think Aloud* [35] é o padrão utilizado para coletar dados durante a avaliação de usabilidade de interfaces de sistemas em geral. Esse consiste na observação de participantes durante a execução de um experimento. Os participantes são orientados a verbalizar

suas espectativas, estratégias, dificuldades e objetivos, além de serem supervisionados por câmeras e gravadores de áudio. Através da verbalização, o pesquisador condutor do experimento, é capaz de verificar situações que evidenciam dificuldades e facilidades encontradas pelo desenvolvedor enquanto realizava determinada tarefa na interface sendo avaliada.

O uso do *Think Aloud* já está consolidado como a metodologia para avaliar usabilidade de interfaces, principalmente interfaces gráficas. Contudo, no contexto de interfaces programáveis, o ambiente difere do ambiente gráfico. Por conta disso, pesquisadores conduziram estudos para adequar o protocolo *Think Aloud* para o ambiente de interfaces programáveis [14]. Essa adequação vem se consolidando como técnica para a avaliação de APIs. Por exemplo, Ellis et al [5] utilizaram o protocolo para comparar a criação de objetos utilizando o padrão Factory e utilizando construtores. Outro exemplo importante, é o estudo realizado por Stylos e Clarke [44] para avaliar o impacto de construtores com parâmetros na usabilidade da API. Para reforçar o uso da adequação, vários outros trabalhos [43] [5] [45] [28] relatam o sucesso da aplicação do protocolo *Think Aloud* para APIs.

O protocolo *Think Aloud For APIs* também requisita que os participantes de um experimento verbalizem suas espectativas, estratégias, dificuldades e objetivos. No contexto de APIs, os participantes devem externar, por exemplo, a dificuldade em encontrar determinado método, ou a facilidade ou dificuldade em mapear um conceito presente no problema para uma abstração presente no código.

Uma vez que essas metodologias de avaliação estão consolidadas, elas foram usadas na avaliação desse trabalho, cuja implementação e resultados estão detalhados no Capítulo 5.

2.6 Estudo Introdutório

Nessa seção será apresentado um estudo introdutório realizado a fim de verificar a realização de testes de sistema para aplicações distribuídas, como esses testes são executados e o conhecimento de ferramentas que auxiliam nessa atividade por parte dos desenvolvedores de sistemas distribuídos. Para isso, foi executado um *survey* [32] entre a comunidade que desenvolve software distribuído.

2.6.1 *Survey*

Existem diferentes tipos de *survey*, sendo a natureza do problema da pesquisa o preponderante a ser levado em consideração na escolha do método de execução. No caso desse estudo introdutório, utiliza-se o *design* descritivo - *case control* [33], pois o objetivo é estudar, através de observações, como as aplicações distribuídas são testadas.

As variáveis de interesse definidas para esse estudo são: (i) realização de testes de sistema para aplicações distribuídas; (ii) modo de execução desses testes (automáticos ou manuais); (iii) e o conhecimento de ferramentas que auxiliam nessa atividade por parte dos desenvolvedores de sistemas distribuídos.

2.6.2 População e Instrumento

A população alvo é constituída pelo conjunto de indivíduos no qual o *survey* pode ser aplicado [34]. Neste estudo, a população alvo foi a comunidade que desenvolve e/ou testa software distribuído ou que já tenha feito pelo menos uma dessas atividades em um passado recente.

O instrumento do *survey*, geralmente um questionário, é o meio pelo qual o pesquisador consegue coletar informações dos respondentes [32]. Devido a sua importância, o instrumento deve ser criado e validado através de técnicas bem definidas, para que não haja ambiguidades ou incompreensões no instrumento, motivando a participação dos respondentes.

As duas maneiras mais comuns de organizar a validação de um instrumento são: grupos focados e estudo piloto [33]. Um grupo focado é composto por pessoas interessadas na pesquisa e *experts* nas áreas envolvidas, os quais devem analisar o instrumento e fornecer seus *feedbacks* ao elaborador do mesmo. Já o estudo piloto é uma versão inicial do instrumento distribuída para um número menor de respondentes com o intuito de verificar se o questionário está sendo compreendido da maneira esperada.

Para o estudo que realizamos, o grupo focado foi formado por 06 pessoas (02 especialistas na área de software distribuído, 02 especialistas em testes, 01 respondente e interessado no resultado da pesquisa e 01 especialista na língua em que o instrumento foi escrito - inglês). O estudo piloto foi realizado com os líderes técnicos de 07 projetos desenvolvidos no Laboratório de Sistemas Distribuídos [17] da Universidade Federal de Campina Grande

[16].

A partir dos *feedbacks* obtidos e da análise das respostas do estudo piloto, o instrumento do nosso *survey* evoluiu até chegar à sua versão final a qual foi enviada para os respondentes e está disponível em https://spreadsheets.google.com/spreadsheet/viewform?formkey=dDVHekRrMXdfajFSRFNlQ0xUUEx6SGc6MA.

2.6.3 Seleção da Amostra e Divulgação do Instrumento

Uma amostra é um subconjunto da população alvo para o qual são estudados os atributos de interesse [29]. E, para este trabalho, a obtenção da amostra seguiu um método amostral não-probabilístico conhecido como bola de neve [34].

Inicialmente, foi realizada uma busca na Internet por projetos de softwares distribuídos. Com isso, encontramos cerca de 60 projetos e seus respectivos responsáveis.

Em seguida, um e-mail contendo o instrumento do *survey* foi enviado ao representante de cada um dos projetos. A mensagem deste e-mail explicava a pesquisa juntamente com seus objetivos, e, pedia que o participante respondesse o instrumento uma vez para cada projeto de software distribuído que está ou esteve sob sua responsabilidade em um passado recente. Além disso, no texto também havia um pedido para que o respondente encaminhasse o questionário para seus colegas de profissão que também pudessem contribuir com a pesquisa. Logo, o tamanho da amostra tende a crescer de acordo com a experiência e a rede de contatos dos participantes.

2.6.4 Resultados e Análise

O *survey* foi enviado a um conjunto de cerca de 60 líderes de projetos de aplicações distribuídas, espalhados geograficamente ao longo das Américas, Europa, Ásia e Oceania, os quais foram selecionados como discutido em 2.6.3. Um total de 22 respostas foi obtido, porém a taxa de resposta dos participantes não pôde ser calculada devido ao método de amostragem utilizado, o bola de neve. As respostas foram analisadas de acordo com as variáveis de interesse para o estudo apresentadas em 2.6.1.

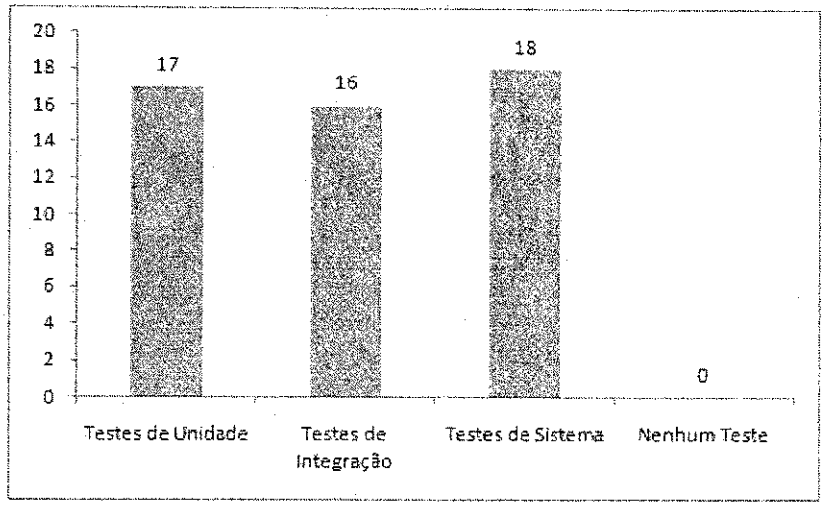

Figura 2.1: Realização de testes

Quanto a realização de testes de sistemas

De acordo com as respostas coletadas, há indícios de que 82% dos projetos que desenvolvem software distribuído realizam testes de sistema. A taxa de realização de testes de sistemas foi superior aos outros tipos de testes questionados, pois 77% dos respondentes realizam testes de unidade e 73% realizam testes de integração.

Todos os participantes do *survey* afirmaram que realizam pelo menos um dentre esses três tipos de teste. Na Figura 2.1 pode ser observado o número de projetos que afirmaram realizar cada um dos tipos de testes.

Quanto a automatização de testes de sistemas

Como pode ser observado na Figura 2.2, dentre os participantes que realizam testes de sistema, apenas 5% tem seus testes completamente automatizados. A maioria deles, 60%, o fazem de maneira parcialmente manual e parcialmente automatizada, sobrando 35% que realizam esse tipo de teste de forma completamente manual. Dentre os participantes que efetuam alguma técnica de automatização, mesmo que apenas parcial, 100% dos respondentes afirmaram que o fazem a partir de *shell scripts*.

Logo, há indícios de que apenas uma pequena parcela dos desenvolvedores realizam testes de sistemas de aplicações distribuídas de maneira completamente automatizada. Bem como também pode se notar que o uso de *scripts* é a principal alternativa na tentativa de

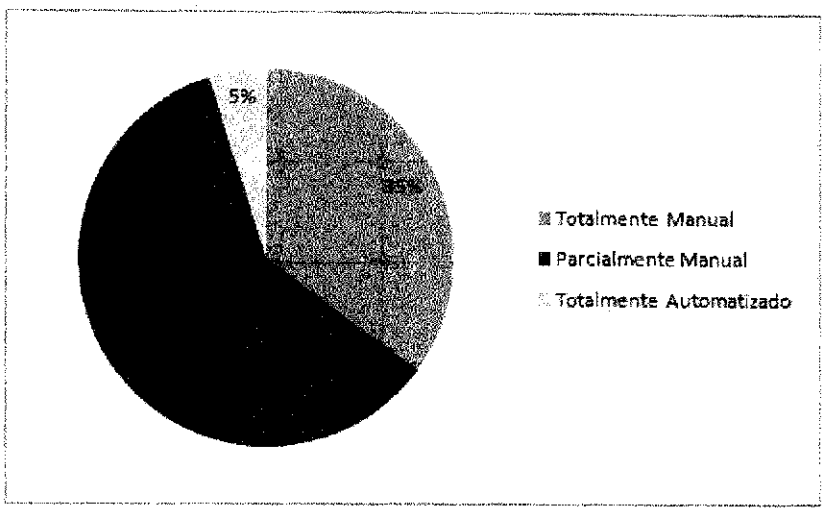

Figura 2.2: Automatização de testes de sistemas

automatizar, mesmo que parcialmente, a execução de testes de sistemas de softwares distribuídos.

Quanto a importância de testes de sistemas

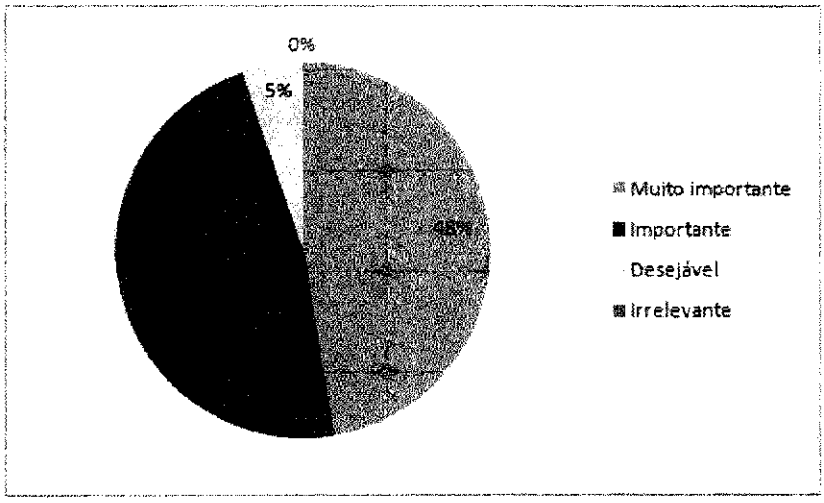

Figura 2.3: Importância de testes de sistemas

Pode-se observar na Figura 2.3 que boa parte dos que realizam testes de sistema em seus projetos, 48%, consideram-nos muito importantes e, outros 47% os consideram "apenas" importantes para assegurar a qualidade do produto desenvolvido. Uma pequena parcela, representada por 5% dos respondentes, classificou esse tipo de teste como desejável. Nenhum dos participantes afirmou que os testes de sistemas são irrelevantes.

Esses números mostram que há indícios de que existe o conhecimento dos benefícios que testes de sistema trazem para um projeto de software distribuído, levando-os a serem classificados com tal importância.

Quanto ao conhecimento de ferramentas de apoio à prática de testes de sistemas

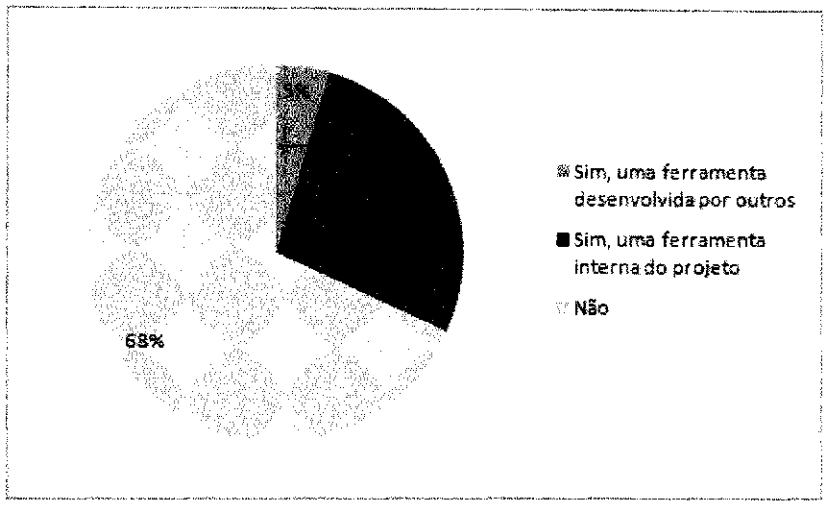

Figura 2.4: Conhecimento sobre ferramentas

Como apresentado na Figura 2.4, quando questionados sobre o conhecimento de ferramentas que facilitem a escrita e execução de testes de sistema, 68% dos respondentes afirmaram desconhecer essas ferramentas. Já no caso de 27% dos participantes, eles sabem de alguma ferramenta interna, ou seja, ferramenta implementada pelo mesmo projeto que desenvolve o software distribuído. Outros 5% conhecem alguma ferramenta externa de apoio à realização do teste, ou seja, ferramenta desenvolvida por outras equipes e/ou projetos e disponibilizadas para uso.

Um conjunto de *scripts* já desenvolvidos e prontos para uso pode ser considerado como uma ferramenta interna ao projeto que auxilie na execução dos testes. No caso dos participantes que afirmam conhecer alguma ferramenta que ajude nessa atividade, mencionaram ferramentas como: SmartFrog [38], BOINC [3], PlanetLab [4] e emulab [6]. Essas ferramentas apóiam em determinado aspecto (principalmente como ferramenta de automatização de *deployment* ou como *testebed*), considerando suas funcinalidades, o programador que quer automatizar testes. Porém, nenhuma delas tem como objetivo principal a automatização de testes. Portanto, as respostas indicam que não há ferramentas conhecidas e/ou divulgadas

cujo objetivo seja proporcionar meios para que aplicações distribuídas sejam testadas automaticamente.

Quanto à relação de variáveis

Neste ponto, deseja-se analisar as possíveis relações existentes entre algumas das variáveis examinadas neste trabalho. Mais especificamente, busca-se alguma evidência de que dois grupos de variáveis estejam ou não relacionadas entre si: (1) se a realização de testes de sistemas de aplicações distribuídas reflete em uma menor taxa de bugs reportados após o lançamento de um novo *release* do software; e (2) se o conhecimento de alguma ferramenta de apoio influencia na realização ou não de testes de sistema. Porém, fazer a análise de relação entre variáveis categóricas é uma tarefa difícil. No entanto, alguns testes foram propostos para verificar se variáveis desse tipo são independentes, como o teste qui-quadrado de Pearson [39] e o teste exato de Fisher [21].

Um teste de independência avalia se os resultados observados em duas variáveis, expressas em uma tabela de contingência, são independentes um do outro. O cálculo de um teste desse tipo produz um p-valor [42]. Então, assumindo-se uma hipótese nula de que as duas variáveis são independentes, esta hipótese nula poderá ser rejeitada a um determinado nível de significância α se o valor obtido para o p-valor for menor que α, caso contrário não será possível rejeitar a hipótese nula para aquele nível de significância.

Em estatística, tabelas de contingência são utilizadas para reunir informações e analisar as relações entre duas ou mais variáveis, que normalmente são variáveis categóricas [31]. Já considerando testes de independência, o teste exato de Fisher [21] é o mais apropriado para esse trabalho por utilizar valores significativamente mais exatos e ser indicado no caso da amostra do trabalho ser considerada pequena.

		Taxa de *bugs* reportados			
		BAIXA	RAZOÁVEL	ALTA	TOTAL
Realização de testes de sistema	SIM	10	7	1	18
	NÃO	1	3	0	4
	TOTAL	11	10	1	22

Tabela 2.1: Realização de Testes de Sistemas X Taxa de Bugs

A Tabela 2.1 representa a tabela de contingência das variáveis: realização de testes de sistemas; e taxa de *bugs* reportados *pós-release*. Nela pode ser observado que 55.5% dos que realizam testes de sistema afirmam que a taxa de *bugs* reportados *pós-release* do software é baixa. Já para 39%, a taxa de *bugs* é razoável, e, para apenas 5.5% dos que realizam esse tipo de teste a taxa de *bugs* reportados é alta. Considerando aqueles que não realizam testes de sistema, a maioria (75%) afirmou que a taxa de *bugs* reportados *pós-release* é razoável.

		Conhecimento de ferramentas		
		SIM	NÃO	TOTAL
Realização de testes de sistema	SIM	5	13	18
	NÃO	2	2	4
	TOTAL	7	15	22

Tabela 2.2: Realização de Testes de Sistemas X Conhecimento de Ferramentas

A Tabela 2.2 representa a tabela de contingência das variáveis: realização de testes de sistemas; e conhecimento de ferramentas de apoio. Nela pode-se observar que 73% dos participantes que realizam testes de sistema não conhecem qualquer ferramenta que facilite a implementação e execução dos testes. No caso dos demais, apenas 27% dos que realizam testes de sistema conhecem alguma ferramenta desse tipo. Contudo, 50% dos que não fazem testes de sistema afirmaram que conhecem alguma ferramenta de apoio.

A linguagem R [40] foi usada para calcular o p-valor para o teste de independência das variáveis apresentadas nas Tabelas de Contingência 2.1 e 2.2. Considerando essas tabelas, os resultados do p-valor foram 0.4361 e 0.5646, respectivamente. Com isso, levando em consideração a mesma hipótese nula, de que as duas variáveis são independentes, para ambas as tabelas e os níveis de significância $\alpha 1$=0.05 e $\alpha 2$=0.10, não há evidências para rejeitar a hipótese nula em nenhuma das tabelas. Pois, os valores dos p-valor de ambas as tabelas são maiores que os níveis de significância considerados.

Portanto, considerando a amostra utilizada para esse estudo e os níveis de significância considerados, não há evidências para afirmar que a realização de testes de sistema relaciona-se com a taxa de *bugs* reportados *pós-release* do software. Também não pode-se afirmar estatisticamente que a realização de testes de sistema relaciona-se com o conhecimento de

alguma ferramenta que auxilie essa atividade. Na seção 2.6.6 é detalhado porque o *survey* motivou a continuação desse trabalho.

2.6.5 Ameaças à validade dos resultados

Alguns argumentos podem ser levantados para ameaçar a validade do estudo realizado através desse *survey*. Essas ameaças são pontuadas abaixo juntamente com os cuidados tomados para diminuir a chance de as mesmas acontecerem.

- **Compreensão das questões por parte dos respondentes:** mesmo que os elaboradores imaginem que as questões estão claras, diferentes respondentes podem entender a mesma questão de diferentes maneiras. Para melhorar o instrumento neste aspecto, o mesmo foi analisado por *experts* da área (grupo focado), foi realizado um estudo piloto antes da aplicação do questionário, como também todos os termos técnicos utilizados no instrumento do *survey* foram definidos no mesmo;

- **Compreensão dos conceitos abordados por parte dos respondentes:** mesmo que os termos técnicos abordados no questinário façam parte do dia a dia dos respondentes, é possível que diferentes participantes tenham definições diferentes para o mesmo termo. Por exemplo: alguns desenvolvedores consideram um teste de sistema mesmo sem o sistema executar em um ambiente similar ao que será submetido quando em produção. Para evitar essa ameaça, todos os termos técnicos utilizados foram definidos no início do questionário;

- **Amostra pode não ser representativa:** esse argumento pode ser mencionado devido ao tamanho da amostra não ter sido tão grande quanto o desejado. Para evitar que a amostra fosse pequena, foi utilizado o método de amostragem bola de neve, onde o tamanho da amostra deve crescer proporcionalmente à experiência e à lista de contatos dos respondentes. E, para que a análise não seja questionada, foi utilizado o teste exato de Fisher para calcular o *p-value*, este é o método destinado a pesquisas com amostras pequenas;

- **Inibição dos desenvolvedores que não realizam testes:** como pode ser observado em 2.6.4, a grande maioria dos respondentes realizam testes de sistema. Contudo, esse

resultado pode ter ocorrido devido a possível não participação dos desenvolvedores que não realizam esse tipo de teste por receio de perder credibilidade e/ou divulgação desses dados. Para diminuir esse risco, em nenhum momento o instrumento requisita informações sobre identificação do software considerado para as respostas nem tampouco identificação sobre o participante;

- **Imprecisão nas respostas:** por existirem perguntas com alternativas de respostas categóricas e, de certa forma, subjetivas, o respondente pode ter escolhido uma das opções sem mensurar com mais precisão a realidade do seu projeto. Isso se deve ao fato de não haver uma escala bem definida para criar categorias distintas para dados dessa natureza. A análise realizada por *experts* da área (grupo focado) e o estudo piloto foram utilizados também com o intuito de melhorar o instrumento para diminuir este risco;

- **Coleta de respostas sobre diferentes realidades:** projetos diferentes possuem complexidades diferentes. E, da mesma forma, apresentarão casos de testes mais ou menos complexos. Por esse motivo, torna-se difícil analisar conjuntamente dados que refletem essas diferentes realidades. Em outras palavras, alguns projetos podem ter mais dificuldades na realização de testes do que outros por tratarem de um produto mais complexo.

2.6.6 Conclusões

De acordo com os resultados obtidos durante esse estudo introdutório, grande parte dos projetos envolvendo softwares distribuídos que participaram do survey realizam testes de sistema. Contudo, a esmagadora maioria dos que fazem esse tipo de teste, fazem-no ou de maneira totalmente manual ou parcialmente automatizada com o uso de *shell scripts*, isto é, não conseguem automatizar a execução de 100% dos casos de teste, mas para o trecho automático fazem uso de *shell script*.

O uso de *scripts* para automatizar testes de sistema implica em ter que usar uma outra linguagem para a implementação do teste, resultando na mudança de ambiente de desenvolvimento. Em vista disso, supõe-se que o uso desse recurso tende a dificultar tanto a implementação e manutenção dos casos de testes durante a evolução do software quanto a legibilidade/clareza dos testes por parte dos integrantes da equipe, que podem não estar

acostumados com esse outro ambiente de desenvolvimento.

O *survey* apresentado apontou que quase 70% dos desenvolvedores participantes da pesquisa não conhecem qualquer ferramenta que possa ajudar na automatização de testes de sistema de softwares distribuídos. No caso de outros 27% dos respondentes, eles conhecem apenas algum meio utilizado pelo próprio projeto na tentativa de automatizar os testes. Já 5% dos participantes afirmam conhecer alguma ferramenta de apoio ao teste disponível ao uso.

As ferramentas mencionadas pelos programadores foram: SmartFrog [38], BOINC [3], PlanetLab [4] e emulab [6]. Entretanto, nenhuma dessas ferramentas tem como foco principal disponibilizar meios para a automatização de testes de sistemas distribuídos. Essas ferramentas apóiam em determinado aspecto, considerando suas funcionalidades, o programador que quer automatizar seus testes. Além das ferramentas mencionadas, existem outras tantas com essa natureza, tais como, Capistrano [12], Metronome [2], ETICS [20] e Bamboo [7].

É possível caracterizar e classificar as dificuldades relacinoadas à atividade de testes de aplicações distribuídas em duas categorias distintas:

1. O controle e manipulação da infraestrutura onde o sistema que será testado será implantado para a sua execução;

2. A manipulação e consulta sobre o comportamento do próprio sistema sob teste (do inglês, SUT - *System Under Test*).

O trabalho discutido nessa dissertação refere-se à segunda categoria. Ou seja, considerando que a infraestrutura esteja montada e configurada para a execução do teste, como também que o SUT já esteja devidamente implantado nessa infraestrutura, a questão está em como possibilitar que o testador desenvolva e execute testes de sistema de maneira automática a partir de sua máquina e usando o mesmo ambiente de desenvolvimento que foi utilizado para desenvolver o SUT. Além disso, mesmo fazendo parte do primeiro problema, este trabalho também precisou disponibilizar um meio para que o testador implantasse sua aplicação. Com isso, torna-se possível a automatização do caso de teste.

Capítulo 3

Uma Abordagem para o Desenvolvimento de Testes de Sistema Automáticos de Aplicações Distribuídas

Como vimos, a automatização completa de testes de sistema não é alcançada em grande parte dos projetos de aplicações distribuídas que foram pesquisados e a maioria dos que tentam automatizar algo fazem uso de *shell scripts*. Nesse capítulo, descrevemos uma abordagem para o desenvolvimento de testes de sistema automáticos de aplicações distribuídas.

A nossa técnica propõe que o testador possa implementar testes de sistema automáticos utilizando o mesmo ambiente de desenvolvimento usado para a implementação do SUT. De forma que, através de agentes de teste, seja possível executar comandos nas máquinas onde o SUT está implantado, como também especificamente nos componentes do SUT. Com isso, o testador poderia manipular e consultar o SUT a partir de sua máquina e de maneira automática.

Inicialmente apresentamos a definição de uma API, características de uma boa API e os requisitos considerados para a API proposta (em 3.1). Em seguida, em 3.2, mostramos a especificação da API proposta com as entidades e operações disponibilizadas. Já em 3.3, mostramos as etapas a serem percorridas para o desenvolvimento dos testes automáticos, e por último, em 3.4, é descrito um caso de uso considerando uma aplicação distribuída fictícia como exemplo e um teste para a mesma onde a nossa abordagem de testes pode ser utilizada.

3.1 Contextualização e Requisitos da API

Para que o testador consiga escrever os testes de sistema automáticos usando o mesmo ambiente de desenvolvimento que o SUT, faz-se necessário uma Interface de Programação de Aplicativos (API). Uma API é um conjunto de abstrações e operações que são disponibilizadas para programadores escreverem seus softwares [10].

Uma API é considerada de boa qualidade quando possui algumas características bem definidas [10], entre elas:

- Facilidade para aprender e memorizar;
- Construção de um código limpo, ou seja, código facilmente compreensível por seus usuários;
- Completude, isto é, fornece as abstrações e operações necessárias aos seus usuários;
- Facilidade para estender, ou seja, a API deve permitir facilmente a adição de novos conceitos e abstrações.

Após examinar casos de testes definidos para as aplicações distribuídas desenvolvidas no Laboratório de Sistemas Distribuídos (LSD) [17] da Universidade Federal de Campina Grande (UFCG) [16], bem como requisitos apontados pelos desenvolvedores dessas aplicações, definimos que a API proposta deve atender aos seguintes requisitos:

- O testador deve poder escrever os testes de sistema automáticos usando sua máquina, e, controlar o SUT que está espalhado pela infraestrutura do teste;
- O testador deve poder escrever os casos de testes automáticos de sistema usando a mesma linguagem de programação utilizada durante o desenvolvimento do SUT;
- O testador deve poder usar o mesmo ambiente de desenvolvimento, com todo o suporte já disponível, que foi utilizado para a implementação do SUT;
- O testador deve poder especificar quais as máquinas que serão utilizadas durante a execução do teste;
- Deve ser permitida a copia de aquivos entre a máquina do testador e as máquinas utilizadas para a execução do SUT durante o teste;

- O testador deve poder executar operações nos componentes do SUT disponibilizadas através de um ponto de acesso aos mesmos;
- O testador deve poder executar comandos nas máquinas participantes do teste como se estivesse utilizando o *prompt* da máquina;
- A execução das operações definidas nessa API deve ser bloqueante, com isso busca-se uma baixa incidência de falsos-positivos no teste por asserções antecipadas.

3.2 Especificação da API proposta

Considerando os requisitos anteriormente mencionados, especificamos uma API para o desenvolvimento de testes de sistema automáticos para aplicações distribuídas. As entidades e operações disponibilizadas por essa API, bem como uma visão arquitetural da solução proposta serão apresentadas nas próximas seções.

3.2.1 Componente do Sistema

Essa entidade representa um componente do SUT. Através dela é possível para o programador especificar os dados de um componente do SUT, como: o nome do componente; o arquivo executável desse componente; e um ponto de acesso ao componente. Este último será detalhado na próxima seção.

Considerando a possibilidade de um mesmo componente do SUT executar em mais de uma máquina, é possível que uma única instância dessa entidade seja implantada em mais de uma máquina do teste. O Código 3.1 apresenta, em pseudocódigo, as operações disponibilizadas por essa entidade, as quais servem basicamente para obter informações sobre o componente do sistema.

Código 3.1: Pseudocódigo da API da entidade componente do sistema

```
ComponenteDoSistema {
    obterNome() : Caractere
    obterCaminhoDoArquivoExecutavel() : Caractere
    obterPontoDeAcesso() : PontoDeAcesso
}
```

3.2.2 Ponto de Acesso ao Componente

O desenvolvedor precisa permitir que o testador exercite o SUT da maneira especificada pelo teste. No contexto de aplicações distribuídas, é comum que uma operação realizada em um dos componentes do SUT altere o estado de outro. Portanto, o programador deve permitir que o testador possa tanto exercitar o primeiro, como também consultar o segundo componente. Para isso, definimos a abstração ponto de acesso ao componente.

O desenvolvedor do SUT deve implementar em seu sistema uma instância dessa entidade para cada componente que será exercitado durante os casos de teste. Esse ponto de acesso disponibiliza as operações de ações e consultas necessárias à escrita dos testes automáticos. Um determinado ponto de acesso sempre estará relacionado com apenas um componente do sistema. O Código 3.2 mostra a operação disponibilizada por essa entidade. Com uma chamada à operação **executar(operação, argumentos)**, o testador pode especificar a operação a ser executada no componente do SUT e os argumentos para a mesma, bem como obter o resultado da execução.

Código 3.2: Pseudocódigo da API da entidade ponto de acesso ao componente do sistema

```
PontoDeAcessoAoComponente {
    executar(operação, argumentos) : Resultado
}
```

3.2.3 Máquina de Teste

Essa entidade representa as máquinas que são utilizadas durante a execução do teste. Como discutido na Seção 2.6.6, o trabalho apresentado nessa dissertação admite que as máquinas a serem usadas para o teste já estão prontas e configuradas. Porém, há a necessidade de representar logicamente essas máquinas para permitir a manipulação das mesmas.

Cada programador pode implementar a abstração de uma máquina da maneira mais conveniente para o seu trabalho. No entanto, todas as possíveis implementações para essa entidade devem disponibilizar, no mínimo, as operações apresentadas no Código 3.3.

Código 3.3: Pseudocódigo da API da entidade Máquina de Teste

```
MaquinaDoTeste {
    executarComando(comando) : Caractere
```

```
    implantarComponente(componenteDoSistema) : Nenhum
    executarComponente(nomeComponente, operação, argumentos) : Resultado
    copiarArquivoParaMaquina(origem, destino) : Nenhum
    copiarArquivoDaMaquina(origem, destino) : Nenhum
    obterNumeroComponentes() : Inteiro
}
```

Dentre as operações mostradas no Código 3.3, destacamos e descutiremos as três seguintes:

- **executarComando(comando).** Essa operação permite que o testador execute um comando na máquina como se estivesse utilizando um terminal da mesma. Além disso, ela retorna a saída da execução do comando;
- **implantarComponente(componente).** Essa operação possibilita que o testador implante determinado componente na máquina específica. A ação dessa operação faz com que o arquivo executável do componente seja copiado para a máquina e o ponto de acesso a esse componente fique apto para ser utilizado;
- **executarComponente(componente, operação, argumentos).** Essa operação permite que o testador exercite os componentes já implantados na máquina. Assim, o testador pode especificar qual o componente será exercitado, qual a operação será executada no componente informado (apenas operações definidas no ponto de acesso deste componente poderão ser executadas), e, os argumentos necessários à execução da operação. Além disso, o retorno da execução é disponibilizado ao testador.

3.2.4 Cenário de Teste

Essa entidade existe para ajudar o testador na definição de cenários de teste. Através de uma instância de Cenário de Teste o testador define as máquinas que participam de um teste. Uma vez criada uma instância dessa entidade, ela pode ser utilizada em vários testes, basta que as máquinas especificadas atendam aos requisitos de execução desses diferentes testes.

No momento de criação de um Cenário de Teste, deve-se iniciar a execução de um Agente de Teste em cada uma das máquinas que serão usadas para o teste. A entidade Agente de

Teste será detalhada na próxima seção. O Código 3.4 apresenta as operações disponibilizadas pela abstração Cenário de Teste.

Código 3.4: Pseudocódigo da API da entidade Cenário de Teste

```
CenarioDeTeste {
    executarComando(maquina, comando) : Caractere
    implantarComponente(maquina, componenteDoSistema) : Nenhum
    executarComponente(maquina, nomeComponente, operação, argumentos) :
        Resultado
    obterNumeroMaquinas() : Inteiro
    adicionarMaquina(maquina) : Nenhum
    removerMaquina(maquina) : Nenhum
    encerrar() : Nenhum
}
```

Considerando as operações apresentadas no Código 3.4, destacamos e descutiremos as seguintes:

- **executarComando, implantarComponente e executarComponente.** Essas operações encapsulam operações disponíveis na entidade Máquina de Teste. Por esse motivo, antes de informar os argumentos necessários à sua execução, é preciso informar em qual das máquinas do Cenário de Teste a operação será executada;
- **adicionarMaquina(maquina).** Essa operação permite que o testador adicione uma nova máquina ao Cenário de Teste, ou seja, inicia a execução do Agente de Teste nessa nova máquina. A partir desse ponto é possível realizar operações relacionadas ao teste em execução na nova máquina;
- **removerMaquina(maquina).** Essa operação deve remover a máquina especificada do Cenário de Teste. Essa remoção faz com que a execução do Agente de Teste e dos pontos de acesso aos componentes implantados na máquina sejam interrompidas. Portanto, a partir desse ponto o programador não poderá executar comandos na máquina ou exercitar os componentes. Porém, a remoção da máquina não acontece fisicamente, ou seja, caso o testador não os tenham encerrado, os processos relacionados aos componentes implantados na máquina informada ainda estarão executanto e possivelmente comunicando-se com o restante do sistema.

- **encerrar()**. Essa operação permite que o testador finalize a execução do Agente de Teste em cada uma das máquinas do Cenário de Teste, bem como dos pontos de acesso aos componentes implantados nas mesmas.

3.2.5 Agente de Teste

Essa entidade é responsável por interligar o caso de teste na máquina do testador com os componentes implantados nas Máquinas do Teste. Uma instância dessa entidade deve executar em cada máquina a ser usada durante o teste. Desta forma a abstração Máquina de Teste conhece o Agente de Teste que executa na máquina física que a entidade representa. O Agente de teste também deve conhecer todos os componentes implantados na máquina onde ele executa. Por conhecer os componentes implantados entende-se que o agente de teste deve conhecer o ponto de acesso à cada componente implantado na mesma máquina. O Agente de Teste também é o responsável por executar comandos requisitados pelo testador na máquina.

O Código 3.5 apresenta as operações disponibilizadas pela entidade Agente de Teste.

Código 3.5: Pseudocódigo da API da entidade Agente de Teste

```
AgenteDeTeste {
    executarComando(comando, timeout, diretórioTrabalho) : Caractere
    executarComponente(nomeComponente, operação, argumentos) : Resultado
    iniciarPontoDeAcesso(componente) : Nenhum
    obterNumeroComponentes() : Inteiro
}
```

Como pode ser visto no Código 3.5, algumas das operações disponibilizadas pela abstração Agente de Teste, tais como **executarComando** e **executarComponente**, são semelhantes à algumas da entidade Máquina de Teste. A ideia é que o uso do Agente de Teste seja transparente ao testador, ou seja, o mesmo não precisa saber de sua existência. Assim, o programador utiliza a abstração Máquina de Teste para executar comandos ou exercitar componentes, e, internamente, a Máquina de Teste usa a abstração Agente de Teste para realizar essas ações. Uma visão geral da arquitetura da solução proposta será apresentada na Seção 3.2.6.

Dentre as operações vistas no Código 3.5, faz-se necessário uma breve discussão sobre as seguintes:

- **executarComando(comando, *timeout*, diretórioTrabalho).** Essa operação, assim como a de mesmo nome da abstração Máquina de Teste, permite que o testador execute um comando na máquina como se estivesse em um terminal da mesma. Porém, essa operação requisita mais argumentos, que serão informados pela abstração Máquina de Teste. A máquina onde o comando será executado é a mesma onde o agente estiver executando;

- **executarComponente(componente, operação, argumentos).** A função dessa operação é a mesma da operação de mesmo nome na entidade Máquina de Teste. Portanto, os valores dos argumentos informados para a abstração Máquina de Teste são repassados para a entidade Agente de Teste;

- **iniciarPontoDeAcesso(componente).** Essa operação tem a função de deixar o ponto de acesso ao componente informado apto para uso. Ou seja, após a execução dessa operação, o testador poderá exercitar o componente informado através das operações presentes no ponto de acesso.

Com a existência de uma API com as características e abstrações supracitadas, acreditamos que o testador conseguirá escrever testes de sistema automáticos para aplicações distribuídas usando a mesma linguagem de programação e o mesmo ambiente utilizados para desenvolver o SUT. Além disso, o testador também poderá continuar utilizando os scripts para o SUT, caso eles já existam, porém esses serão chamados a partir de uma de um cenário de teste implementado em uma linguagem de programação. Um estudo de caso implementado em Java para prova de conceito será apresentado no Capítulo 4.

3.2.6 Visão Arquitetural

Uma visão geral da arquitetura da solução proposta é apresentada na Figura 3.1.

Como pode ser visto na Figura 3.1, o testador escreve casos de testes automáticos de sistema utilizando a API de teste proposta nesse trabalho e ferramentas de suporte já disponíveis para o ambiente de desenvolvimento, que é o mesmo que foi utilizado para a implementação do SUT. Assim, o programador consegue fazer uso do suporte já existente para esse ambiente de desenvolvimento bem como do *know how* adquirido durante a implementação do SUT. Além disso, o testador escreve os testes automáticos em sua própria máquina.

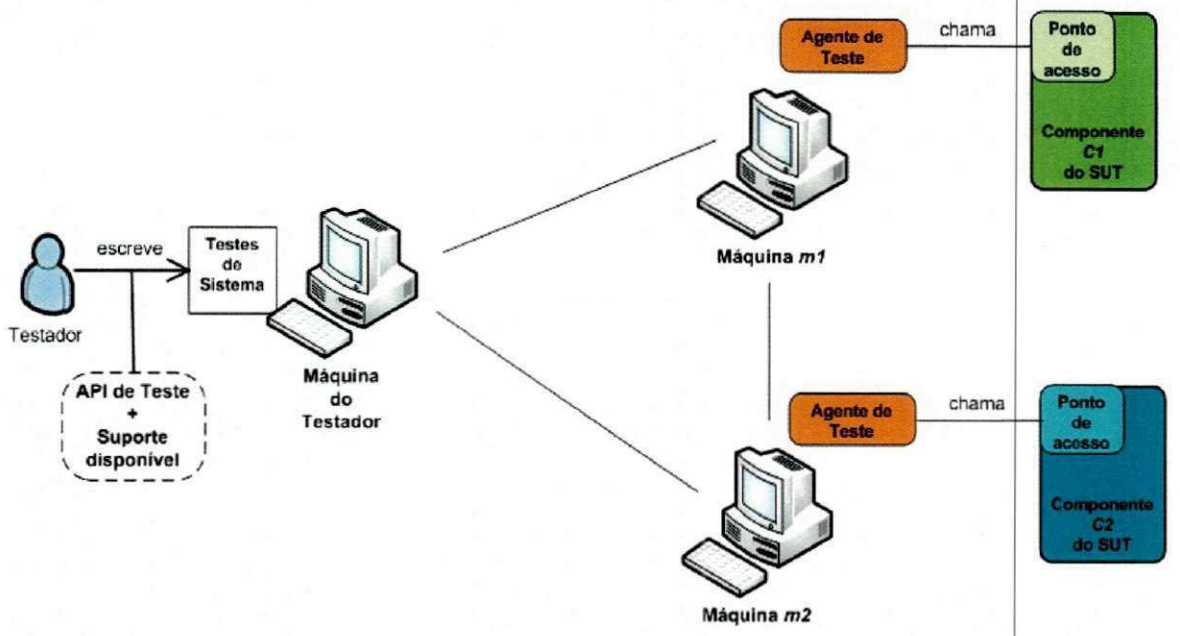

Figura 3.1: Arquitetura da solução proposta

O testador faz uso da abstração Máquina de Teste para representar as máquinas que serão usadas durante o teste. No caso da Figura 3.1, o testador definirá duas instâncias da entidade Máquina de Teste para representar as máquinas *m1* e *m2*. Como já foi mencionado anteriormente, este trabalho admite que a infraestrutura de teste já está montada e configurada para ser usada.

No instante em que o testador cria uma instância da entidade Cenário de Teste informando as duas máquinas que serão usadas durante o teste, um Agente de Teste deve ser iniciado em cada uma das máquinas. O Agente de Teste é o processo, que executa em uma Máquina de Teste, responsável em lidar tanto com a execução de comandos como também com os componentes do SUT implantados na máquina onde ele executa. O testador lida diretamente com a abstração Máquina de Teste, porém, internamente, a entidade Máquina de Teste realiza as operações requisitadas através do Agente de Teste que executa na máquina remota.

Cada componente do SUT, representado pela abstração Componente do Sistema, que será acionado ou consultado durante os testes, precisa ter um ponto de acesso definido. Através desse ponto de acesso, desenvolvido pelo programador do SUT, o testador manipulará e consultará a aplicação de acordo com o que for especificado pelo teste. Na Figura 3.1 o SUT possui dois componentes definidos, *C1* e *C2*, que estão implantados nas máquinas

m1 e *m2* respectivamente. Como também pode ser observado na mesma figura, ambos os componentes possuem um ponto de acesso definido que será conhecido pelo Agente de Teste que executa na máquina onde o componente foi implantado.

As execuções das operações disponibilizadas pelas entidades definidas são bloqueantes. Essa estratégia possibilita que as operações retornem valores como o resultado da operação, bem como busca-se evitar a existência de falsos positivos para asserções antecipadas, já que a execução do teste é bloqueada até que a execução da operação seja encerrada. Desta forma a abordagem proposta põe em prática o padrão de teste *Humble Object* [36]. A utilização desse padrão faz com que toda a lógica de componentes difíceis de testar por diversos motivos (dentre eles a existência de chamadas assíncronas) seja extraída para abstrações que possam ser testadas via testes síncronos, ou seja, com chamadas à metodos bloqueantes. Na abordagem apresentada, essa característica é posta em prática quando define-se um ponto de acesso para cada componente do SUT que precise ser manipulado ou consultado. Então, mesmo que as chamadas ao SUT sejam assíncronas, as chamadas que o agente de teste fará no ponto de acesso aos componentes precisam ser síncronas.

Portanto, a ideia da arquitetura proposta é que o testador consiga representar e definir, em sua máquina e usando um ambiente de desenvolvimento já conhecido, as máquinas que serão usadas durante o teste bem como os componentes do SUT. Cada componente que será exercitado durante o teste deve possuir um ponto de acesso para consulta e/ou realização de ações, bem como deve poder ser implantado nas máquinas. Cada abstração de máquina conhece o Agente de Teste que executa remotamente na máquina real. Através desses agentes que comandos são executados na máquina real como também os componentes do SUT implantados na referida máquina são manipulados.

3.3 Etapas para o desenvolvimento dos testes

Nessa seção apresentamos quais as etapas a serem superadas para que o programador possa escrever testes de sistema automáticos de acordo com a abordagem proposta nesse trabalho.

3.3.1 Preparar o SUT para ser testável

Para tornar o SUT testável, o programador precisa disponibilizar um ponto de acesso aos componentes que serão exercitados durante os testes. Com isso o testador poderá realizar ações e consultar o estado dos componentes ao longo da execução dos testes. Esse ponto de acesso deve ter apenas métodos que permitam que o testador escreva os testes, não havendo necessidade de adicionar operações que não serão utilizadas durante a execução dos casos de testes.

3.3.2 Montar e configurar a infraestrutura que será utilizada durante os testes

Antes de iniciar o teste do software, é necessário que todo o ambiente de execução do caso de teste esteja montado e configurado. Por um ambiente montado, entende-se que as máquinas existem e estão disponíveis para uso, além de sua disposição obedecer à topologia exigida pelo caso de teste (por exemplo: rede local, NAT, ponto a ponto, etc). Já no caso de um ambiente configurado, entende-se que todos os pré-requisitos para a execução do SUT estejam devidamente satisfeitos (por exemplo: sistema operacional, instalação de algum programa específico, etc).

3.3.3 Iniciar o Agente de Teste nas máquinas

Definidas e configuradas as máquinas que serão utilizadas como infraestrutura para a execução dos testes, o próximo passo é iniciar a execução de um agente de teste em cada uma delas. As funcionalidades e características dessa entidade foram discutidas na Seção 3.2.5. O testador só conseguirá executar algum comando ou implantar e executar algum componente nas máquinas após iniciar a execução do Agente de Teste nas mesmas. Pois, o esse agente é encarregado de controlar e acessar tanto a máquina onde está executando como também os componentes do SUT que estejam na mesma. Contudo, ele fará essas tarefa seguindo as requisições definidas pelo testador através da escrita de teste automático.

3.3.4 Escrever os testes de sistema automáticos

Nessa etapa o desenvolvedor deve escrever o teste de sistema para a aplicação distribuída utilizando o mesmo ambiente de desenvolvimento usada para implementar a mesma. Com isso, o desenvolvedor/testador poderá fazer uso do *know how* adquirido durante a fase de desenvolvimento do SUT. Dessa forma, é possível, caso elas existam, utilizar algumas ferramentas que já estejam disponíveis nesse ambiente de desenvolvimento e possam apoiar a realização de testes.

3.4 Exemplo de um Caso de Uso da API

Nessa seção apresentamos o exemplo de uma aplicação distribuída, bem como um possível caso de teste para a mesma. Em seguida mostramos, em pseudocódigos, como o desenvolvedor faria para testar o software através da técnica apresentada.

3.4.1 Aplicação Distribuída

Tomamos como exemplo uma aplicação de mensagens instântaneas. Com o uso da Internet, esse tipo de aplicação é bastante comum hoje em dia.

Na nossa aplicação fictícia existem dois tipos de componentes. O componente usuário é o que representa os usuários da aplicação, ou seja, as pessoas cadastradas que podem utilizar esse sistema. Um usuário só pode trocar mensagens com outros usuários que já estejam adicionados em sua lista de contatos. Para adicionar o usuário *user1* na lista de amigos do usuário *user2* basta chamar a operação que adiciona novos amigos e informar o *login* do contato que será acrescentado. No contexto desse exemplo, quando um usuário adiciona um novo amigo à sua lista, automaticamente e sem a necessidade de confirmação, ele também é adicionado na lista desse novo contato. O mesmo acontece com a remoção de um amigo da lista de contatos, não é necessário confirmação.

O componente servidor é o processo responsável por tratar dos dados de todos os clientes. É nesse componente onde, por exemplo: o *login* e senha dos usuários são verificados; é possível consultar quantos usuários cadastrados existem; e, quantos desses estão conectados em determinado momento.

3.4.2 Caso de Teste

Considerando a aplicação fictícia apresentada na subseção anterior, definimos um possível caso de teste de sistema. O cenário do teste é montado com um componente servidor e dois componentes usuários.

O passo a passo do caso de teste definido é:

1. Iniciar o servidor em uma das máquinas;
2. Criar o usuário *user1* em uma máquina diferente do servidor e verificar se a quantidade de usuários cadastrados é igual a 1, como também que não há usuários conectados nesse instante;
3. Conectar *user1* ao servidor e verificar se a quantidade de usuários conectados é 1;
4. Criar um outro usuário *user2* na terceira máquina e verificar se nesse momento o servidor tem 2 usuários cadastrados e 1 conectado;
5. Conectar *user2* ao servidor e verificar se a quantidade de usuários cadastrados e conectados é 2;
6. Verificar se *user1* e *user2* têm 0 usuários em suas respectivas listas de contatos;
7. Adicionar *user1* à lista de contatos de *user2* e verifica se ambos tem 1 usuário cadastrado em suas respectivas listas de contatos;
8. Encerrar a execução dos usuários e do servidor.

Tendo-se o caso de teste já definido, o desenvolvedor/testador precisa seguir as etapas discutidas na Seção 3.3 para que sua aplicação possa ser testada através da abordagem proposta.

O primeiro passo é criar um ponto de acesso a cada componente do software distribuído que necessitará ser manipulado e/ou consultado durante o caso de teste. No contexto do teste definido, ambos os componentes - servidor e usuário - serão manipulados e consultados. Nesse caso faz-se necessário um ponto de acesso para cada componente. Em pseudocódigos, são apresentados nos Códigos 3.6 e 3.7 como poderiam ser os pontos de acesso aos componentes servidor e usuário respectivamente.

Código 3.6: Pseudocódigo para Ponto de Acesso ao Componente Servidor

```
PontoAcessoServidor{
    iniciarComponenteServidor{
        Inicia o componente servidor da aplicação
    }
    getNumeroUsuariosCadastrados{
        Retorna o número de usuários cadastrados no servidor iniciado
    }
    getNumeroUsuariosConectados{
        Retorna o número de usuários conectados no servidor
    }
}
```

Código 3.7: Pseudocódigo para Ponto de Acesso ao Componente Usuário

```
PontoAcessoUsuario{
    criarComponenteUsuario(nome, senha){
        Cria um componente usuário com o nome e senha informados
    }
    conectarUsuarioAoServidor(nomeDoServidor){
        Conecta o usuário criado ao servidor informado
    }
    getNumeroAmigos(){
        Retorna o número de amigos cadastrados na lista de contatos do
            usuário criado
    }
    adicionaNovoAmigo(loginAmigo){
        Adiciona o usuário com login informado na lista de constatos do
            usuário
    }
}
```

Em seguida deve-se considerar a infraestrutura onde o teste executará. No contexto do teste definido, 3 máquinas são necessárias. Pois existem um servidor e dois usuários e cada um deles deve executar em máquinas diferentes. O próximo passo é iniciar um agente de teste em cada uma das três máquinas que serão utilizadas. Isso deve acontecer quando o testador criar uma instância da entidade Caso de Teste e definir as máquinas que serão usada

durante o teste.

Por último, o testador deve implementar, em sua máquina e usando o mesmo ambiente de desenvolvimento utilizado pelo SUT, o caso de teste definido. No caso definido, o programador poderá manipular e consultar o SUT, que está executando em três máquinas diferentes, a partir de sua máquina e utilizando um ambiente com o qual ele já está acostumado. O Código 3.8 mostra, em pseudocódigo, como o caso de teste definido poderia ser implementado.

Código 3.8: Pseudocódigo para o Caso de Teste

```
CasoDeTesteExemplo {
    / / Montando o cenário
    MáquinaDeTeste mServidor = nova MáquinaDeTeste ( propriedades )
    MáquinaDeTeste mUsuario1 = nova MáquinaDeTeste ( propriedades )
    MáquinaDeTeste mUsuario2 = nova MáquinaDeTeste ( propriedades )
    CenarioDeTeste cenarioTeste = novo CenarioDeTeste ( mServidor ,
        mUsuario1 , mUsuario2 )
    / / Criando os componentes
    servidor = novo ComponenteDoSistema ( servidor , pontoAcessoServidor )
    usuario = novo ComponenteDoSistema ( cliente , pontoAcessoUsuario )
    / / Implantando os componentes nas máquinas
    mServidor.implantarComponente ( servidor )
    mUsuario1.implantarComponente ( usuario )
    mUsuario2.implantarComponente ( usuario )
    / / Iniciando Servidor e verificando
    mServidor.executarComponente( servidor , iniciarComponenteServidor )
    assert( 0 , mServidor.executarComponente( servidor ,
        getNumeroUsuariosCadastrados )
    / / Cria o usuário com nome João senha 12345 em mUsuario1
    mUsuario1.executarComponente( usuario , criarComponenteUsuario, João ,
        12345 )
    assert( 1 , mServidor.executarComponente( servidor ,
        getNumeroUsuariosCadastrados )
    / / Conecta o usuário com nome João senha 12345 em mUsuario1
    mUsuario1.executarComponente( usuario , conectarUsuarioAoServidor ,
        servidorID )
    assert( 1 , mServidor.executarComponente( servidor ,
        getNumeroUsuariosConectados )
    / / Cria o usuário com nome Maria senha 54321 em mUsuario2
```

```
    mUsuario2.executarComponente( usuario , criarComponenteUsuario , Maria ,
        54321 )
    assert( 2 , mServidor.executarComponente( servidor ,
        ggetNumeroUsuariosCadastrados )
    assert( 1 , mServidor.executarComponente( servidor ,
        getNumeroUsuariosConectados )
    / / Conecta o usuário com nome Maria senha 54321 em mUsuario2
    mUsuario2.executarComponente( usuario , conectarUsuarioAoServidor ,
        servidorID )
    assert( 2 , mServidor.executarComponente( servidor ,
        getNumeroUsuariosConectados )
    / / Verifica o número de amigos na lista
    assert( 0 , mUsuario1.executarComponente( usuario , getNumeroAmigos )
    assert( 0 , mUsuario2.executarComponente( usuario , getNumeroAmigos )
    / / usuário1 adiciona usuário2 em sua lista
    mUsuario1.executarComponente( cliente , adicionarNovoAmigo , Maria )
    / / Verifica o número de amigos na lista
    assert( 1 , mUsuario1.executarComponente( usuario , getNumeroAmigos )
    assert( 1 , mUsuario2.executarComponente( usuario , getNumeroAmigos )
}
```

É importante destacar que a fase de montagem de cenário apresentada no Código 3.8, onde há a criação de máquinas, é para a representação lógica das máquinas que serão utilizadas durante o teste. Pois como já foi discutido anteriormente, as máquinas reais precisam estar prontas para uso antes do desenvolvimento dos testes automáticos. Também é notável a semelhança entre o código apresentado e o código produzido para testes de unidade automáticos, que em algumas linguagens já tem um arcabouço de suporte para checagens e publicação de resultados dos testes.

Capítulo 4

SysTest: Um estudo de caso em Java

Nesse capítulo apresentamos um aplicativo, intitulado SysTest, que foi utilizado nos experimentos realizados para demonstrar a viabilidade do uso da abordagem de testes discutida no Capítulo 3. O SysTest é um aplicativo desenvolvido em Java que disponibiliza uma API para apoiar a escrita de testes automáticos de aplicações distribuídas escritas em Java.

As abstrações e operações providas pelo SysTest foram elaboradas levando em consideração os requisitos e o objetivo da API especificada na Seção 3.2. Portanto, com o uso do SysTest o programador deve poder escrever testes automáticos para sua aplicação distribuída a partir de sua máquina e usando o mesmo ambiente de desenvolvimento usado para a implementação do SUT.

É importante destacar que o SysTest não é uma ferramenta pronta para ser posta em produção, pois o mesmo é um estudo de caso implementado em Java para demostrar a viabilidade de pôr em prática a abordagem de teste proposta. Além disso, apesar de ter sido testado, não houve uma equipe de desenvolvimento colaborando com a evolução deste aplicativo.

Então, o SysTest será apresentado nas próximas seções em etapas:

1. **Como possibilitar ao SUT ser testável:** Quais as abstrações e/ou operações disponibilizadas para auxiliar o desenvolvedor a tornar seu SUT testável;

2. **Como lidar com a infraestrutura do teste:** Quais as abstrações e/ou operações disponibilizadas para apoiar o desenvolvedor no relacionamento com as máquinas utilizadas durante o teste;

3. **Agentes de Teste:** Quais as abstrações e/ou operações relacionadas com os agentes de teste e como eles comunicam-se;

4. **Visão Geral:** Como essas diferentes partes do SysTest relacionam-se e comunicam-se.

4.1 Como possibilitar ao SUT ser testável

Como visto anteriormente na Seção 3.3, a primeira etapa para testar um software distribuído é deixá-lo testável. Para isso, deve existir um ponto de acesso aos componentes que serão exercitados e/ou consultados durante os testes. O SysTest disponibiliza um meio para que o desenvolvedor crie esse ponto de acesso aos componentes: a classe *TestableImpl*. Como pode ser observado na Figura 4.1, esta classe implementa a interface *Testable* também definida no SysTest.

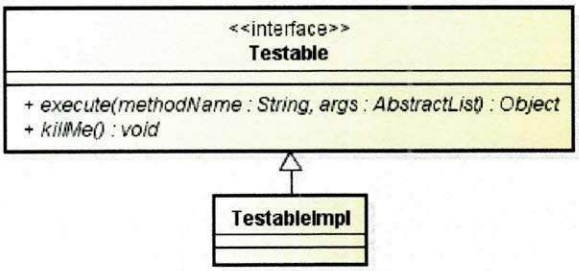

Figura 4.1: Diagrama de Classes do pacote *Testable SUT*

A classe *TestableImpl* implementa os métodos definidos na interface *Testable*: **execute(method, args)** e **killMe()**. O primeiro tem a função de executar o método especificado considerando os argumentos informados. O método a ser executado deve ser definido no ponto de acesso do componente. Já o segundo método, tem a função de encerrar a execução do ponto de acesso ao componente.

A fim de servir como ponto de acesso, o programador do SUT deve desenvolver uma classe intitulada de *TestableClass*. Na *TestableClass* o desenvolvedor deve disponibilizar todas as operações necessárias à escrita dos casos de testes automáticos. A *TestableClass* dos componentes deve estender a classe *TestableImpl*. Deste modo, essa classe desenvolvida pelo programador do SUT será encarada como o ponto de acesso ao componente do SUT relacionado.

A classe *SystemComponent* definida no SysTest representa a entidade Componente do Sistema discutida na Seção 3.2.1. No instante em que um objeto *SystemComponent* é implantado em uma das máquinas do teste, um objeto *TestableClass* (o ponto de acesso ao componente) é instanciado e publicado. Isso não significa que este componente do SUT tenha iniciado sua execução, mas sim que a partir deste ponto existe um objeto publicado que servirá como ponto de acesso ao componente específico. O objeto mencionado é publicado usando a tecnologia *Remote Method Invocation* (RMI) [22] do Java.

4.2 Como lidar com a infraestrutura do teste

A segunda etapa para a escrita de testes automáticos é a montagem e configuração da infraestrutura de execução do teste. O SysTest considera que o ambiente de execução do teste já está totalmente montado e configurado para a correta execução do SUT. Contudo, nessa seção são apresentadas duas abstrações que permitem ao testador lidar com a infraestrutura utilizada para a execução do teste: ***Machine*** e ***TestEnvironment***.

Cada programador pode implementar a abstração de uma máquina da maneira mais conveninte para o seu problema. Para isso, basta fazer sua abstração (classe) implementar a interface ***Machine*** definida no SysTest. Essa abstração definida no SysTest representa a entidade Máquina de Teste discutida na Seção 3.2.3.

O SysTest também disponibiliza uma classe que representa Máquina de Teste padrão e implementa a interface *Machine*: a classe ***DefaultMachine***. Um objeto desse tipo pode ser instanciado usando um arquivo de propriedades como o apresentado na Figura 4.2. Mais detalhes sobre as propriedades que podem ser especificadas no arquivo bem como o significado de cada uma delas podem ser vistos no Apêndice A.

A entidade Cenário de Teste definida na Seção 3.2.4 é implementada pela classe ***TestEnvironment*** disponibilizada pelo SysTest. No momento de criação de um objeto desse tipo, o testador informa uma lista de *Machines*. Essas representam as máquinas que serão utilizadas durante o teste. Nesse momento será iniciado o agente do SysTest em cada uma das máquinas informadas sem que o usuário requisite. Portanto, a partir desse momento será possível executar comandos e implantar componentes do SUT nessas máquinas.

Como apresentado na Seção 3.2.4, também é possível que o testador adicione ou re-

```
surubim.properties
#Configurações do teste
hostname=surubim.lsd.ufcg.edu.br
path_to_deploy=/local/giovanni/
#In miliseconds
timeout=5000
jar_file_path=/home/giovanni/systest.jar
test_agent_log=surubim.out
test_agent_err_log=surubim.err
```

Figura 4.2: Exemplo de um arquivo de propriedades para um objeto *DefaultMachine*

mova máquinas do caso de teste. Quando adiciona-se uma máquina, um agente do SysTest é iniciado na mesma. Da mesma forma que quando se remove uma máquina do caso de teste, a execução do agente é encerrada. É importante observar que, quando remove-se uma máquina, apenas a execução do agente e dos objetos *Testable Classes* publicados são encerrados. Portanto, não mais será possível executar comandos ou exercitar os componentes na máquina.

4.3 Agentes de Teste

Definidas as máquinas participantes do teste, um agente do SysTest deve ser iniciado em cada uma delas. Como discutido na Seção 3.2.5, é através desses agentes que o testador executará operações nas máquinas e nos componentes do SUT. O SysTest disponibiliza algumas abstrações para lidar com o Agente de Teste e elas são apresentadas na Figura 4.3.

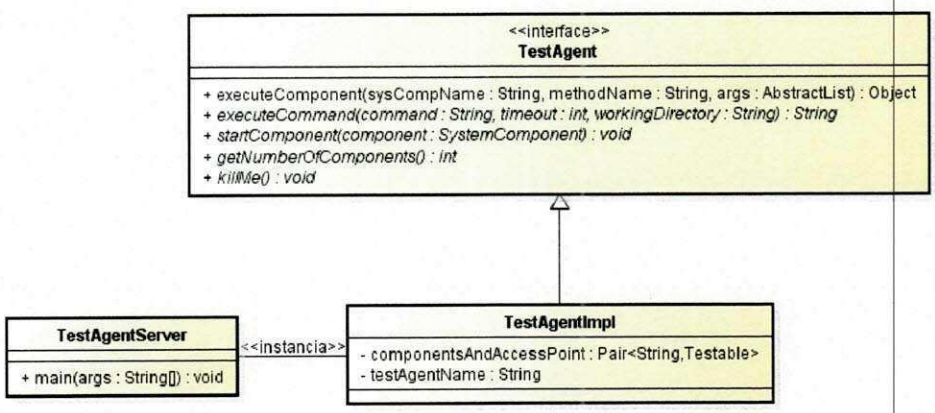

Figura 4.3: Diagrama de classes do pacote *TestAgent*

A interface ***TestAgent*** representa a entidade Agente de Teste discutida na Seção 3.2.5.

As operações especificadas para essa entidade Agente de Teste são definidas na interface em questão, porém é a classe ***TestAgentImpl*** que implementa os métodos definidos em *TestAgent*. A tecnologia Java *Remote Method Invocation* (RMI) [22] foi utilizada para a comunicação remota entre a máquina do testador e os agentes de teste executando nas máquinas de teste, como também entre os agentes e os objetos do tipo *TestableImpl* que representam o ponto de acesso aos componentes. A classe ***TestAgentServer*** é chamada no momento de instanciação do *TestAgentImpl*, ou seja, no instante em que é preciso iniciar a execução do agente de teste. Pois é através da primeira que um objeto da segunda é iniciado e publicado usando Java RMI.

4.4 Visão geral

A Figura 4.4 apresenta uma visão geral do SysTest de acordo com a arquitetura da solução proposta.

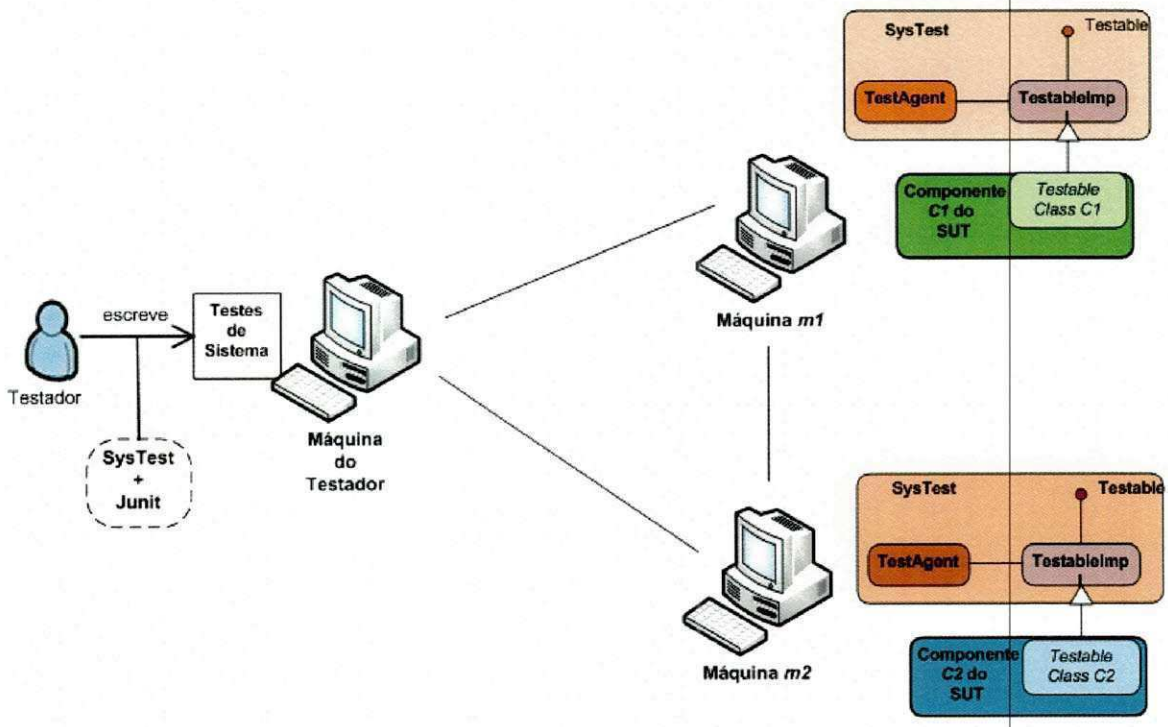

Figura 4.4: Arquitetura do SysTest

Como pode ser visto na Figura 4.4, o testador escreve casos de testes automáticos de sistema utilizando a API do SysTest e o arcabouço de suporte disponibilizado pelo *framework*

JUnit [18], principalmente na fase de checagem e publicação dos resultados dos testes. O teste é desenvolvido na máquina do testador e utiliza o mesmo ambiente de desenvolvimento usado para implementar o SUT, como por exemplo a IDE (*Integrated Development Environment*) Eclipse [19] ou Netbeans [37]. É preciso que o SysTest esteja disponível em todas as máquinas participantes do teste como também na máquina do testador. Assim, é possível que o programador use as abstrações definidas pelo aplicativo para escrever o teste automático.

No teste automático (na máquina do testador), serão definidos objetos *Machine*, *TestEnvironment* e *SystemComponent* para manipular a execução do teste. Enquanto que, nas máquinas de teste, acontece a execução de um objeto *TestAgent* da classe *TestAgentImpl* em cada uma delas. O objeto *TestAgent* armazena informações sobre os componentes do SUT implantados na máquina onde ele executa. Deste modo, como pode ser observado na Figura 4.4, o *TestAgent* relaciona-se com a classe *TestableImpl*, pois esta representa o ponto de acesso aos componentes. Portanto, para cada componente implantado na máquina, o *TestAgent* possui um *TestableImpl* relacionado. A classe *TestableImpl* implementa os métodos definidos na interface *Testable*.

O desenvolvedor do SUT precisa implementar um ponto de acesso aos componentes que serão exercitados durante os testes. Com o uso do SysTest, o programador deve implementar a *TestableClass* do componente. Essa classe deve estender a classe *TestableImpl*, assim será considerada o ponto de acesso ao componente em questão. Na Figura 4.4 pode-se verificar que existem dois componentes do SUT, *c1* e *c2*, implantados nas máquinas *m1* e *m2* respectivamente, e, cada um deles possui uma *TestableClass* desenvolvida pelo testador.

As execuções dos métodos disponibilizados pelas abstrações definidas no SysTest são bloqueantes. Essa estratégia possibilita que as operações retornem valores como o resultado da operação, bem como busca-se evitar a existência de falsos positivos por asserções feitas em momentos indequados, já que a execução do teste é bloqueada até que o fluxo de execução retorne da chamada ao método. Porém, mesmo assim não é garantido que falsos positivos não acontecerão. Pois o testador pode chamar um método bloqueante na abstração *Machine*, que por sua vez chama um método bloqueante no *TestAgent*, que também chama um método bloqueante na *TestableClass* de um componente, que pode chamar métodos assíncronos no componente do SUT. Assim, é possível que o fluxo de execução volte para o teste sem que a execução da operação anterior no componente do SUT tenha sido completamente finalizada.

Portanto, quando o testador deseja executar determinada operação em um dos componentes do SUT, ele requisita essa ação a abstração *Machine*. Essa abstração internamente aciona o *TestAgent* que executa remotamente na máquina real. O agente identifica qual dos componentes na máquina será exercitado, e aciona o *TestableImpl* do mesmo. Por último, o *TestableImpl* executa o método especificado na *TestableClass* do componente em questão.

Capítulo 5

Avaliação

Nesse capítulo apresentamos a metodologia de avaliação sobre a viabilidade de uso da abordagem de testes discutida no Capítulo 3. Para prova de conceito, desenvolvemos um estudo de caso do aplicativo que auxilia o desenvolvimento de testes de sistema de aplicações distribuídas intitulada de SysTest. Esse estudo de caso foi desenvolvido em sua totalidade usando a linguagem de programação Java. Com a implementação do SysTest concluída, desenvolvemos uma aplicação distribuída fictícia para escrevermos testes de sistema para a mesma usando o SysTest. Uma vez avaliado através de uma aplicação fictícia, realizamos um experimento com desenvolvedores de dois sistemas distribuídos reais e em produção para avaliarmos a usabilidade do SysTest.

A estratégia de demonstração de viabilidade da abordagem de teste proposta nesse trabalho é apresentada como segue: em 5.1, mostramos o uso do SysTest para desenvolver testes de sistema para uma aplicação fictícia; e em seguida, em 5.2 apresentamos o estudo realizado sobre a usabilidade da API do SysTest de acordo com a opinião de desenvolvedores que o utilizaram para escrever testes para uma aplicação real e em produção.

5.1 Uso do SysTest para testar uma aplicação fictícia

A fim de fazer uso do SysTest, desenvolvemos uma aplicação distribuída fictícia. Com isso, foi possível escrever casos de testes automáticos para esse software e verificar, sob nosso ponto de vista, a viabilidade de escrever testes de sistema automáticos usando as abstrações e operações do SysTest.

A aplicação fictícia desenvolvida para esta etapa foi a aplicação de mensagens instantaneas discutida na Seção 3.4. O caso de teste escrito também foi o discutido na mesma seção.

No momento inicial, fizemos o papel do desenvolvedor do SUT, ou seja, implementamos as classes necessárias ao funcionamento da aplicação fictícia. Detalhes de implementação, bem como o código dessas classes podem ser observados no Apêndice C. Em um segundo momento, utilizamos o SysTest para escrever o teste automático realizando as implementações necessárias para o uso do SysTest bem como obedecendo as etapas apresentadas em 3.3.

5.1.1 Implementação das *Testable Classes*

Como a aplicação fictícia tem dois componentes e ambos serão exercitados durante o teste, cada um deles precisa de uma *TestableClass*. Os Códigos 5.1 e 5.2 são os códigos produzidos para as *Testable Classes* do componente servidor e usuário respectivamente.

Código 5.1: *Testable Class* do Componente Servidor

```
public class TestableServerClass extends TestableImpl {
   Server server;
   String serverID;

   public TestableServerClass() throws RemoteException {
      super();
   }

   public String startServer(String serverName) throws RemoteException,
      MalformedURLException, UnknownHostException{
      server = new ServerImpl();
      serverID = "rmi://" + InetAddress.getLocalHost().getHostName() + "
         :1099/" + serverName;
      Naming.rebind(serverID, server);
      return serverID;
   }

   public int getNumberOfRegisteredUsers() throws RemoteException{
      return server.getNumberOfRegisteredUsers();
```

```
    }

    public int getNumberOfConnectedUsers() throws RemoteException{
        return server.getNumberOfConnectedUsers();
    }

    public void stopServer() throws RemoteException, MalformedURLException
        , NotBoundException{
        Naming.unbind(serverID);
    }
}
```

Código 5.2: *Testable Class* do Componente Usuário

```
public class TestableUserClass extends TestableImpl{
    User user;
    String userID;

    public TestableUserClass() throws RemoteException {
        super();
    }

    public String registerUser(String name, String password, String
        serverID) throws RemoteException, MalformedURLException,
        UnknownHostException, NotBoundException{
        userID = "rmi://" + InetAddress.getLocalHost().getHostName() + "
            :1099/" + name;
        user = new UserImpl(name, password, serverID, userID);
        Naming.rebind(userID, user);
        return userID;
    }

    public boolean connectToServer() throws MalformedURLException,
        RemoteException, NotBoundException{
        return user.connectToServer();
    }

    public int getNumberOfFriends() throws RemoteException{
```

```
        return user.getNumberOfFriends();
    }

    public boolean addFriend(String otherUserID) throws RemoteException,
        MalformedURLException, NotBoundException{
        return user.addFriend(otherUserID);
    }

    public void stopUser() throws RemoteException, MalformedURLException,
        NotBoundException {
        Naming.unbind(userID);
    }
}
```

Como pode ser observado nos Códigos 5.1 e 5.2, a *TestableClass* de um componente deve estender a classe *TestableImpl* definida no SysTest. Além disso, é importante frisar que, mesmo que um componente do SUT tenha mais ações e funcionalidades disponíveis, sua *TestableClass* deve conter apenas as operações necessárias ao testador para a escrita dos testes definidos. Nesse contexto, as Figuras 5.1 e 5.2 possuem as operações necessárias à implementação do caso de teste definido em 3.4. No entanto, quanto mais funcionalidades do sistema os testes explorarem, possivelmente mais operações declaradas nas *Testable Classes* existirão.

5.1.2 Implementação do caso de teste definido

Definidas as *Testable Classes* dos componentes do SUT, o próximo passo foi escrever o teste para o caso definido. O Código 5.3 mostra o caso de teste definido escrito usando o SysTest, o JUnit [18] e as *Testable Classes* implementadas. A classe desenvolvida possui 3 métodos: o *setUp()*, o *tearDown()* e o *testCase1()*. Antes dos métodos, da linha 2 a 4, são definidos os objetos que serão utilizados ao longo da classe. Pode-se notar que o código produzido para o teste de sistema é similar ao código produzido quando escrevendo testes de unidade com o *framework* JUnit [18].

Código 5.3: Classe de Teste da Aplicação Fictícia usando o SysTest

```
1 public class ToyExampleTest {
```

```
2     static Machine mServer, mUser1, mUser2;
3     static TestEnvironment test;
4     static String serverID, user1ID, user2ID;
5
6     @BeforeClass
7     public static void setUp() throws Exception {
8        //Creating machines
9        mUser2 = new DefaultMachine("/home/giovanni/toyExample/mandarim.
            properties");
10       mServer = new DefaultMachine("/home/giovanni/toyExample/tubarao.
            properties");
11       mUser1 = new DefaultMachine("/home/giovanni/toyExample/tilapia.
            properties");
12       List<Machine> machines = new LinkedList<Machine>();
13       machines.add(mServer);
14       machines.add(mUser1);
15       machines.add(mUser2);
16       //Creating the testEnvironment with specified machines
17       test = new TestEnvironment(machines);
18       //Creating components of the SUT
19       SystemComponent server = new SystemComponent("server", "/home/
            giovanni/toyExample/toyExample.jar", "TestableServerClass");
20       SystemComponent user = new SystemComponent("user", "/home/giovanni
            /toyExample/toyExample.jar", "TestableUserClass");
21       //Deploying components of the SUT
22       mServer.deployComponent(server);
23       mUser1.deployComponent(user);
24       mUser2.deployComponent(user);
25    }
26
27    @AfterClass
28    public static void tearDown() throws Exception {
29       mUser1.executeComponent("user", "stopUser");
30       mUser2.executeComponent("user", "stopUser");
31       mServer.executeComponent("server", "stopServer");
32       test.reset();
33    }
```

```
34
35    @Test
36    public void testCase1() throws Exception{
37       serverID = (String)mServer.executeComponent("server", "startServer"
             , "ExampleServer");
38       assertEquals(0, (Integer)mServer.executeComponent("server", "
             getNumberOfRegisteredUsers"));
39       assertEquals(0, (Integer)mServer.executeComponent("server", "
             getNumberOfConnectedUsers"));
40
41       user1ID = (String)mUser1.executeComponent("user", "registerUser", "
             João", "123456", serverID);
42       assertEquals(1, (Integer)mServer.executeComponent("server", "
             getNumberOfRegisteredUsers"));
43       assertEquals(0, (Integer)mServer.executeComponent("server", "
             getNumberOfConnectedUsers"));
44
45       user2ID = (String)mUser2.executeComponent("user", "registerUser", "
             Maria", "654321", serverID);
46       assertEquals(2, (Integer)mServer.executeComponent("server", "
             getNumberOfRegisteredUsers"));
47       assertEquals(0, (Integer)mServer.executeComponent("server", "
             getNumberOfConnectedUsers"));
48       //Connecting to the server
49       assertTrue((Boolean) mUser1.executeComponent("user", "
             connectToServer"));
50       assertTrue((Boolean) mUser2.executeComponent("user", "
             connectToServer"));
51       //Check connection
52       assertEquals(2, (Integer)mServer.executeComponent("server", "
             getNumberOfRegisteredUsers"));
53       assertEquals(2, (Integer)mServer.executeComponent("server", "
             getNumberOfConnectedUsers"));
54       //Check the number of friends
55       assertEquals(0, (Integer)mUser1.executeComponent("user", "
             getNumberOfFriends"));
56       assertEquals(0, (Integer)mUser2.executeComponent("user", "
```

```
            getNumberOfFriends"));
57          //Adding friend
58          assertTrue((Boolean) mUser1.executeComponent("user", "addFriend",
                user2ID));
59          assertEquals(1, (Integer)mUser1.executeComponent("user", "
                getNumberOfFriends"));
60          assertEquals(1, (Integer)mUser2.executeComponent("user", "
                getNumberOfFriends"));
61      }
62  }
```

Considerando o Código 5.3, O método *setUp()* (linha 7 a 25) é executado antes de todos os métodos da classe. Isso se deve ao uso da *annotation @BeforeClass*, que é definida pelo JUnit para essa função. É nesse trecho que o cenário do teste é montado. Primeiramente, da linha 9 a 11, as três máquinas necessárias ao teste são instanciadas, cada uma considerando o arquivo de propriedades informado. Na linha 17 é inicializado um objeto *TestEnvironment*, o qual recebe como argumento as máquinas que participarão do teste. Em seguida, nas linhas 19 e 20, são instanciados dois objetos *SystemComponent*. Esses objetos representam os componentes do SUT, e, no momento de criação requisitam: um nome pelo qual será conhecido, seu arquivo *.jar* e o nome de sua *TestableClass*. Por último, foi realizada a implantação dos componentes nas máquinas.

Devido a *annotation @AfterClass*, o método *tearDown()* (linhas 28 a 33) é executado depois de todos os outros métodos. Nele estamos encerrando a execução dos componentes do SUT através dos métodos: *stopUser* e *stopServer*. Esse encerramento é realizado de acordo com o que está implementado nos métodos, no caso da nossa aplicação fictícia, como pode ser visto nos Códigos 5.1 e 5.2, esses métodos fazem um *unbind* dos objetos *user* e *server* respectivamente. Em seguida o método *reset* do objeto *TestCase* é chamado. Nesse contexto, esse método irá encerrar a execução dos agentes de teste que executam nas máquinas.

O caso de teste em si está implementado no método *testCase1()* (linhas 36 a 61). O primeiro passo é iniciar o componente servidor, ação que é feita através do método *startServer* presente na *Testable Class* do servidor. Como pode ser ovservado na linha 37 do Código 5.3, faz-se a inicialização do servidor com o nome *ExampleServerName* na máquina *mServer*. Em seguida, nas linhas 38 e 39, ocorre a verificação se o servidor inicializado não possui

nenhum usuário cadastrado ou conectado.

Na linha 41, foi criado um usuário com nome João e *password* igual a 123456, como também o cadastro dele no servidor informado. Novamente, nas linhas 42 e 43, verifica-se o número de usuários cadastrados e conectados ao servidor. Porém, desta vez é esperado que exista 1 usuário registrado. Na linha 45, o usuário de nome Maria e *password* 654321 é criado e registrado. Após esse ponto, ocorre a verificação se existem 2 usuários registrados e 0 conectados.

O próximo passo do teste foi conectar os usuários criados ao servidor (linhas 49 e 50). Já nas linhas 52 e 53, tem-se a verificação se exitem 2 usuários tanto registrados como conectados ao servidor respectivamente.

Como os dois usuários foram criados nesse instante, ambos não possuem amigos em sua lista de contatos, esse fato é checado nas linhas 55 e 56. Logo após, na linha 58, o usuário de *mUser1* adiciona o segundo usuário à sua lista de contatos. De acordo com as regras da aplicação fictícia discutidas em 3.4, o usuário da máquina *mUser2* também adiciona o primeiro usuário automaticamente. Por fim, nas linhas 59 e 60, ocorre a verificação se cada usuário tem apenas um amigo em sua lista de contatos.

Execução do Teste Implementado

A Figura 5.1 mostra a execução do teste automático escrito usando o SysTest e JUnit. Como se nota na mesma, utilizando o mesmo ambiente de desenvolvimento, o programador pode utilizar o *know how* adquirido durante a implementação do SUT, bem como todas as funcionalidades que o ambiente já oferece. O JUnit 4 é utilizado para as checagens do teste (através de seus *asserts*) e publicação dos resultados (barras verde, quando não houver falhas nos testes, ou vermelha quando pelo menos um dos testes falhar). Além disso, como pode ser visto na Figura 5.1 o código produzido é semelhante ao código gerado por testes de unidade usando o JUnit. É importante destacar que o desenvolvedor deve prestar muita atenção no momento da escrita de teste, pois o resultado da execução pode ser barra vermelha por conta de algum erro de implementação do teste.

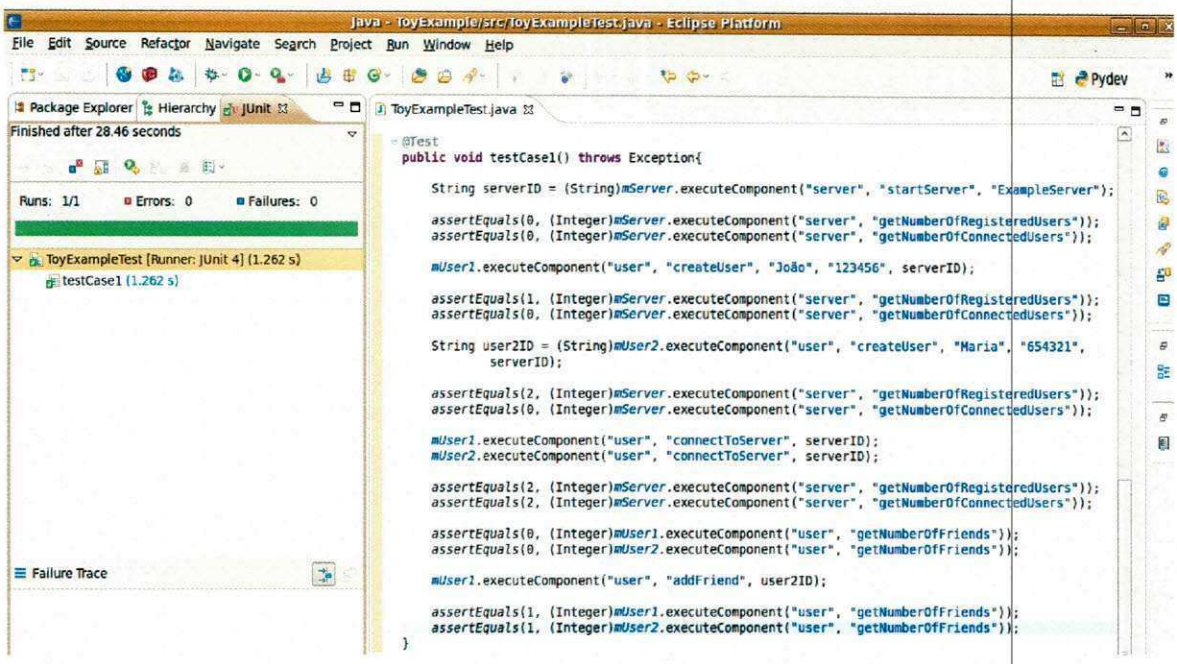

Figura 5.1: Execução do Caso de Teste

5.1.3 Outros casos de testes para a aplicação fictícia

Além do caso de teste tomado como exemplo e discutido em 3.4, nós escrevemos outros casos de testes automáticos para a nossa aplicação distribuída fictícia. Esses serão apresentados nas próximas seções.

Teste envio de mensagem entre usuários

O Código 5.9 apresenta a implementação do caso de teste para verificar se o usuário implantado em *mUser1* consegue enviar uma mensagem para o usuário em *mUser2* e vice-versa.

Código 5.4: Teste de Envio de Mensagem entre Usuários

```
1 @Test
2 public void testCase2() throws Exception {
3     //Checking last messages
4     assertEquals("", (String)mUser1.executeComponent("user", "
          getLastReceivedMessage"));
5     assertEquals("", (String)mUser1.executeComponent("user", "
          getLastSentMessage"));
6     assertEquals("", (String)mUser2.executeComponent("user", "
          getLastReceivedMessage"));
```

```
7     assertEquals("", (String)mUser2.executeComponent("user", "
          getLastSentMessage"));
8     //user1 sends message
9     String message1 = "Hello user 2!";
10    assertTrue((Boolean)mUser1.executeComponent("user", "sendMessage",
          message1, user2ID));
11    //Checking last messages
12    assertEquals("", (String)mUser1.executeComponent("user", "
          getLastReceivedMessage"));
13    assertEquals(message1, (String)mUser1.executeComponent("user", "
          getLastSentMessage"));
14    assertEquals(message1, (String)mUser2.executeComponent("user", "
          getLastReceivedMessage"));
15    assertEquals("", (String)mUser2.executeComponent("user", "
          getLastSentMessage"));
16    //user2 sends message
17    String message2 = "Hi user 1! How are you?";
18    assertTrue((Boolean)mUser2.executeComponent("user", "sendMessage",
          message2, user1ID));
19    //Checking last messages
20    assertEquals(message2, (String)mUser1.executeComponent("user", "
          getLastReceivedMessage"));
21    assertEquals(message1, (String)mUser1.executeComponent("user", "
          getLastSentMessage"));
22    assertEquals(message1, (String)mUser2.executeComponent("user", "
          getLastReceivedMessage"));
23    assertEquals(message2, (String)mUser2.executeComponent("user", "
          getLastSentMessage"));
24 }
```

Inicialmente, das linhas 3 a 7, ocorre a verificação se os usuários nem enviaram nem receberam mensagens até esse momento. Para isso, verificou-se a última mensagem recebida e enviada de ambos se são iguais a uma mensagem vazia. Na linha 10, houve a verificação se o *user1* enviou a mensagem para *user2* sem problemas. Feito isso, da linha 12 a 15, houve a checagem se a última mensagem enviada por *user1* e a última mensagem recebida por *user2* são iguais, como também se não houve alterações nas mensagens recebidas por *user1*

e enviadas por *user2*.

Em seguida, na linha 18, o *user2* envia uma mensagem para o *user1*. E por último, das linhas 20 a 23, verifica-se se as últimas mensagens enviadas e recebidas de ambos os usuários estão de acordo com o esperado.

A escrita desse caso de teste e dos que serão apresentados nas próximas subseções requer operações que não estão disponibilizadas na *Testable Class* do componente usuário apresentada no Código 5.2. Portanto, a Testable Class de um componente pode evoluir, e, isso será discutido na seção 5.1.4.

Teste de envio de mensagem para desconhecido

Esse caso de teste deve verificar que um usuário não conseguirá enviar uma mensagem para um usuário que não esteja em sua lista de contatos. O Código 5.5 mostra a implementação desse caso de teste. Na linha 4 do Código 5.5 ocorre a checagem se quando o *user1* tenta enviar mensagem para um usuário desconhecido, o retorno da operação é falso, ou seja, a mensagem não é enviada com sucesso.

Código 5.5: Teste de Envio de Mensagem para Usuário Desconhecido

```
1 @Test
2 public void testCase3() throws Exception {
3     String unkonwUserID = "unknonUser";
4     assertFalse((Boolean)mUser1.executeComponent("user", "sendMessage", "
          Hello user2!", unknonwUserID));
5 }
```

Teste de remoção de contato da lista de um usuário

Nesse caso de teste checamos se um usuário consegue remover um dos contatos de sua lista de contatos. O Código 5.6 expõe a implementação desse teste. Nas linhas 4 e 5, verifica-se se ambos os usuários possuem 01 contato em suas respectivas listas. Já na linha 7 o *user1* remove o *user2* de sua lista e ocorre a checagem se foi possível realizar essa operação. Como especificado em 3.4, quando um usuário remove outro de sua lista, automaticamente o segundo também deve excluir o primeiro. Portanto, nas linhas 9 e 10, verifica-se se realmente a remoção ocorreu e os dois usuário possuem 0 contatos em sua lista de contatos.

Código 5.6: Teste de Remoção de Contato da Lista de um Usuário

```
1  @Test
2  public void testCase4() throws Exception {
3      //Checking number of friends
4      assertEquals(1, (Integer)mUser1.executeComponent("user", "
           getNumberOfFriends"));
5      assertEquals(1, (Integer)mUser2.executeComponent("user", "
           getNumberOfFriends"));
6      //Removing friend
7      assertTrue((Boolean) mUser1.executeComponent("user", "removeFriend",
           user2ID));
8      //Checking number of friends
9      assertEquals(0, (Integer)mUser1.executeComponent("user", "
           getNumberOfFriends"));
10     assertEquals(0, (Integer)mUser2.executeComponent("user", "
           getNumberOfFriends"));
11 }
```

Teste de desconexão de usuário

A verificação de desconexão resume-se a checar se o usuário consegue se desconectar do servidor. O Código 5.7 apresenta a implementação desse teste. Nas linhas 4 e 5 se verificou se existem 2 usuários registrados e conectados ao servidor. Na linha 7 o *user1* pede para se desconectar do servidor e ocorre a checagem se ele conseguiu. Nesse ponto deve haver 2 usuários registrados e 1 conectado, isso é verificado nas linhas 9 e 10. Já na linha 12, o *user2* repete o pedido feito pelo *user1* na linha 7. E, por último, nas linhas 14 e 15, checa-se se existem 2 usuários registrados e 0 conectados.

Código 5.7: Teste de Desconexão de um Usuário

```
1  @Test
2  public void testCase5() throws Exception {
3      //Checking number of registered and connected user
4      assertEquals(2, (Integer)mServer.executeComponent("server", "
           getNumberOfRegisteredUsers"));
5      assertEquals(2, (Integer)mServer.executeComponent("server", "
           getNumberOfConnectedUsers"));
```

```
6     //Disconnecting user 1
7     assertTrue((Boolean)mUser1.executeComponent("user", "disconnect"));
8     //Checking number of registered and connected user
9     assertEquals(2, (Integer)mServer.executeComponent("server", "
          getNumberOfRegisteredUsers"));
10    assertEquals(1, (Integer)mServer.executeComponent("server", "
          getNumberOfConnectedUsers"));
11    //Disconnecting user2
12    assertTrue((Boolean) mUser2.executeComponent("user", "disconnect"));
13    //Checking number of registered and connected user
14    assertEquals(2, (Integer)mServer.executeComponent("server", "
          getNumberOfRegisteredUsers"));
15    assertEquals(0, (Integer)mServer.executeComponent("server", "
          getNumberOfConnectedUsers"));
16 }
```

Teste de cancelamento do registro de usuário

Esse caso de teste serve para verificar se o usuário efetua sem problemas o cancelamento de seu registro no servidor. A implementação desse teste pode ser observada no Código 5.8. Na linha 4 do referido código verifica-se se existem 2 usuários registrados no servidor. Em seguida, na linha 6, o *user1* pede para se desregistrar. Nesse ponto deve existir apenas 1 usuário cadastrado, o que é checado na linha 8. Por último, o *user2* também pede para se desregistrar (linha 10) e verifica-se se não existem mais usuários cadastrados no servidor (linha 12).

Código 5.8: Teste de Cancelamento do Registro de Usuário

```
1  @Test
2  public void testCase6() throws Exception {
3     //Checking number of registered users
4     assertEquals(2, (Integer)mServer.executeComponent("server", "
          getNumberOfRegisteredUsers"));
5     //Unregistering user
6     mUser1.executeComponent("user", "unregister");
7     //Checking number of registered users
```

```
8     assertEquals(1, (Integer)mServer.executeComponent("server", "
          getNumberOfRegisteredUsers"));
9     //Unregistering user
10    mUser2.executeComponent("user", "unregister");
11    //Checking number of registered users
12    assertEquals(0, (Integer)mServer.executeComponent("server", "
          getNumberOfRegisteredUsers"));
13 }
```

5.1.4 Evolução das *Testable Classes* dos componentes do SUT

Nota-se que para a escrita desses novos casos de teste, faz-se necessária a escrita de operações que a *TestableClass* do componente usuário apresentado no Código 5.2 ainda não possui. Tomando como exemplo o teste de envio de mensagem, as operações que faltam são: *getLastReceivedMessage*, *getLastSentMessage* e *sendMessage*. Logo, para a correta execução automática do teste considerado, o trecho que deve ser adicionado no Código 5.2 é mostrado no Código 5.9.

Código 5.9: Métodos ausentes na *TestableClass* para testar envio de menssagem

```
1  public String getLastReceivedMessage() throws RemoteException{
2     return user.getLastReceivedMessage();
3  }
4
5  public String getLastSentMessage() throws RemoteException{
6     return user.getLastSentMessage();
7  }
8
9  public boolean sendMessage(String message, String userID) throws
       RemoteException{
10    return user.sendMessage(message, userID);
11 }
```

À medida que cresce o número de funcionalidades do sistema que os testes exploram, cresce também o número de operações necessárias nas *Testable Classes* dos componentes. Pois, quanto mais funcionalidades explorar, mais operações serão necessárias aos testadores para escrever o caso de teste. No entanto, é interessante observar que tais métodos são apenas

delegadores, sendo bastante simples de serem implementados ou até de ter sua implementação automatizada por um *plugin* de uma IDE como por exemplo o eclipse [19].

Para permitir a escrita dos outros testes definidos em 5.1.3 as *Testable Classes* apresentadas nos Códigos 5.1 e 5.2 precisaram evoluir. A versão final das *Testable Classes* desse exemplo que faz uso de uma aplicação fictícia, bem como mais detalhes de implementação podem ser vistos no Apêndice C. A Figura 5.2 mostra a execução de todos os casos de teste definidos e implementados para a aplicação fictícia.

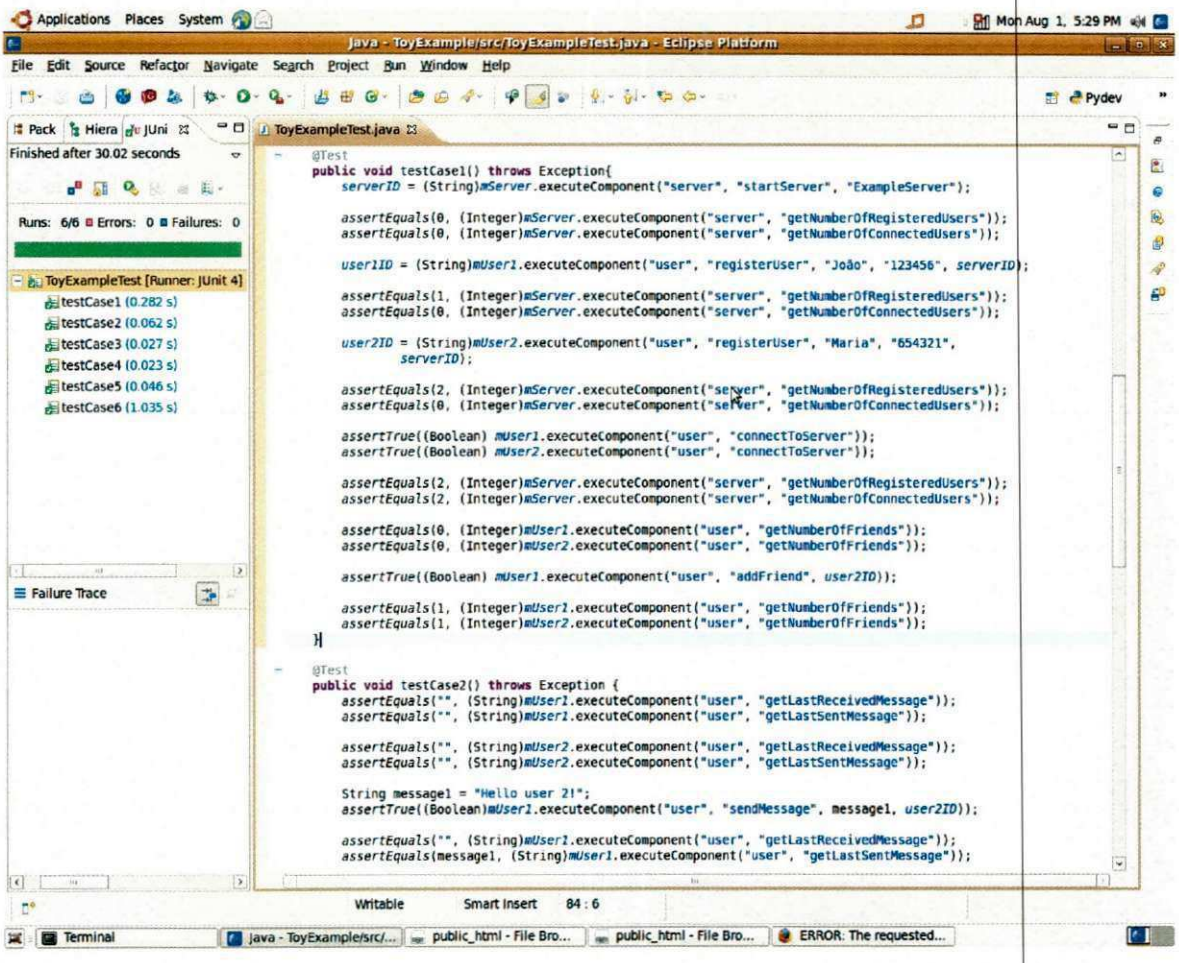

Figura 5.2: Execução dos casos de teste definidos

5.1.5 Conclusões

Esse estudo serviu para mostrar que é possível desenvolver diferentes testes automáticos para aplicações distribuídas usando o SysTest e o mesmo ambiente de desenvolvimento do SUT. Os exemplos de testes apresentados na seção anterior consideram que as alterações

no cenário de teste realizadas pelos testes anteriores são mantidas. Com o uso do SysTest, também é possível isolar o cenário para cada caso de teste. Para isso, é necessário montar as configurações necessárias antes das asserções serem realizadas.

O desenvolvimento da aplicação fictícia apresentada, bem como dos casos de testes definidos foi de extrema importância para identificar quais as necessidades que os testadores têm para a realização desses testes. Através do uso do SysTest, pudemos identificar em quais aspectos o mesmo poderia evoluir com o inuito de melhorar a legibilidade e simplicidade dos testes automáticos. Porém, o aplicativo desenvolvido e utilizado como exemplo inicial não representa um software real com suas necessidades e características. Por isso, a fim de avaliar a usabilidade do SysTest no desenvolvimento de testes de sistema automáticos para aplicações reais e em produção, foi realizada a avaliação apresentada na proxima seção.

5.2 Usabilidade da API do SysTest

5.2.1 Metodologia

Visão Geral

De acordo com Steven Clarke [14], na avaliação de usabilidade de uma API, o interessante é fazer a comparação entre o que os desenvolvedores esperam e o que a API provê. Considerando isso, a metodologia utilizada consistiu em uma breve palestra introdutória sobre o tema testes de sistema de aplicações distribuídas e os requisitos e objetivos da API, para, em seguida, o desenvolvedor definir um caso de teste de sistema da sua aplicação para implementá-lo usando a API.

Inicialmente, requisitou-se que os programadores desenvolvessem o caso de teste definido através de pseudocódigo. Nesse contexto, o pseudocódigo representa o que os desenvolvedores esperam que exista na API para que eles possam escrever os testes. Após essa primeira fase, foi indicado que os desenvolvedores implementassem o mesmo caso de teste utilizando a API do SysTest e o JUnit, ferramenta esta já conhecida pelos desenvolvedores. Todas essas tarefas foram realizadas na máquina de trabalho do participante, ou seja, espaço no qual o ambiente de desenvolvimento do SUT já está pronto e configurado. Os participan-

tes tinham acesso à Internet e à documentação do SysTest [1]. Durante essa etapa, foi utilizado o protocolo *Think Aloud For APIs* [14] para a coleta de dados. Esse protocolo consiste em estimular os programadores a verbalizarem suas espectativas, estratégias, dificuldades e objetivos ao utilizar a API sendo observada.

Os desenvolvedores foram monitorados por uma câmera de vídeo e tiveram seus discursos gravados através do uso de um microfone. Por fim, anotações a respeito do comportamento dos desenvolvedores foram feitas pelo condutor do experimento. Todas essas ferramentas foram utilizadas com o objetivo de coletar uma grande quantidade de dados relevantes à avaliação da API.

Ao final do experimento, os participantes foram requisitados a responder um questionário[2] que também foi usado como fonte de dados para a análise. Esse questionário foi elaborado baseado no *Framework* de Dimensões Congnitivas proposto por Green e Petre [26] e adaptado por Clarke [15].

Fases

A metodologia foi dividida em seis etapas:

1. Definição dos Projetos a Serem Utilizados;
2. Escolha dos Participantes;
3. Palestra Introdutória Sobre Testes de Sistema e Requisitos;
4. Escolha dos Casos de Testes a Serem Implementados;
5. Execução do Experimento;
6. Análise dos Dados.

[1] A documentação do SysTest pode ser acessada no endereço eletrônico http://www.lsd.ufcg.edu.br/giovanni/doc/

[2] Esse questionário foi aplicado via *web* e está disponível no endereço eletrônico https://spreadsheets.google.com/spreadsheet/viewform?formkey=dDh6SjhRYmVwcGU0bGwwWkk2S2VIT1E6MQ. Além disso também pode ser observado no Apêndice B dessa dissertação.

Etapa 1. Definição dos Projetos a serem Utilizados no Experimento

As aplicações distribuídas escolhidas para participarem do experimento foram: o OurGrid [47] e o BeeFS [11]. O OurGrid é um *middleware* para um ambiente de grade computacional aberta, escrito em sua totalidade na linguagem Java e trata-se de um sistema em produção desde 2004. Já o BeeFS é um sistema de arquivos distribuído que agrega o espaço ocioso dos discos de estações de trabalho de uma rede local. Também é escrito totalmente em Java e está em produção desde 2009.

Além das características já mencionadas, a escolha destas aplicações para o experimento é decorrente também de outros fatores, como por exemplo, o fato de suas equipes de desenvolvimento estarem localizadas no mesmo laboratório em que este trabalho foi desenvolvido, facilitanto o contato com os coordenadores e desenvolvedores dos projetos. Além disso, ambos possuem casos de testes de sistema bem definidos que não são realizados de maneira automática por falta de mecanismos viáveis que o façam.

Etapa 2. Escolha dos Participantes

Foram convidados a participar do experimento todos os componentes da equipe de desenvolvimento do OurGrid e do BeeFS. A experiência mínima de um ano com a linguagem de programação Java foi considerada como ponte de corte dos participantes. Essa estratégia mirou impedir a participação de programadores inexperientes, pois seria complicado determinar se o participante estava com dificuldade em utilizar a API do SysTest ou com a própria linguagem Java. Os participantes serão melhor caracterizados quando tratarmos da análise das respostas do questionário em 5.2.2.

Etapa 3. Palestra Introdutória Sobre Testes de Sistema e Requisitos

Nessa etapa apresentamos uma breve introdução sobre o tema. Antes do experimento, realizamos uma palestra de no máximo 30 minutos para cada um dos participantes individualmente. Essa palestra abordou alguns conceitos, tais como, testes, testes de sistema, testes de sistema de softwares distribuídos e automatização de testes. Um resumo dos resultados do estudo introdutório, descrito em 2.6, também foi apresentado. Além disso, foram expostos o objetivo geral e os requisitos básicos da API que seria avaliada. Por último, foi apresentado

um exemplo de teste de acordo com a abordagem proposta nesse trabalho, considerando um aplicativo exemplo e um caso de teste para o mesmo. Durante essa palestra não foi usado código da API, ou seja, quando necessário, pseudocódigos foram utilizados.

Etapa 4. Escolha dos Casos de Testes a serem Implementados

Essa etapa da metodologia foi realizada pelos participantes do experimento. Após a palestra, foi requisitado a cada um dos participantes que definisse um caso de teste de sistema da aplicação que ele desenvolve para ser implementado usando a API do SysTest. Esses casos de testes poderiam já existir e serem executados como também poderia ser um caso de teste para o SUT que ainda não fosse executado. Essa estratégia consiste, portanto, no desenvolvimento testes reais das aplicações, ou seja, testes que surgiram das necessidades reais dos projetos, os quais são relevantes ao projeto sob o ponto de vista de seus desenvolvedores. Os casos de testes implementados serão melhor caracterizados quando tratarmos da análise das respostas do questionário em 5.2.2.

Etapa 5. Execução do Experimento

O experimento foi executado no Laboratório de Sistemas Distribuídos (LSD) [17] da Universidade Federal de Campina Grande (UFCG) [16]. Cada participante, individualmente, escreveu o teste de sistema escolhido para sua aplicação. Em um primeiro momento, essa escrita foi feita em pseudocódigo. Após essa etapa, os participantes tiveram que escrever o mesmo teste utilizando a API do SysTest e o JUnit. Durante essa fase os participantes puderam consultar apenas a documentação do SysTest e a Internet como fonte de informações, ficando o aplicador do experimento impossibilitado de sanar quaisquer dúvidas com relação ao SysTest.

Foi utilizado o protocolo *Think Aloud* para colher dados a respeito das expectativas dos participantes em relação à API do SysTest. E por último, os participantes foram submetidos a um questionário que abordava questões relacionadas ao caso de teste implementado e a usabilidade da API.

Etapa 6. Análise dos Dados

Com o término da execução do experimento e da coleta dos dados, quantificamos e estudamos os dados para então começar a etapa de análise dos mesmos. Essa análise foi dividida em três fases:

1. Comparação entre pseudocódigos e a API do SysTest
2. Análise dos dados oriundos do protocolo *Think Aloud*
3. Análise das respostas do questionário

Comparação entre pseudocódigos e a API do SysTest

Essa comparação foi realizada com o objetivo de verificar se a API atendia às expectativas dos desenvolvedores. Isso foi feito através da análise entre as intenções do usuário nos pseudocódigos e a API do SysTest.

Análise dos dados oriundos das observações

Nessa etapa, a avaliação do áudio e das anotações feitas durante o exeperimento a ser usado o protocolo *Think Aloud* se mostraram de suma importância para fazer o embasamento da análise dos resultados. Já com relação aos vídeos com o comportamento dos programadores, poucos dados foram avaliados. Na análise de interface gráfica o vídeo como fonte de dados é mais interessante. No entanto quando se trata de APIs, as outras formas se mostram mais precisas.

Análise dos dados do questionário

Tanto para a elaboração do questionário como para a análise dos dados coletados pelo mesmo, usou-se como guia o *Framework* de Dimensões Cognitivas proposto por Green e Petre [26] e adaptado por Clarke [15].

O *Framework* de Dimensões Cognitivas descreve 12 dimensões que individualmente e coletivamente causam impacto tanto no aprendizado de uma API como na maneira como seus usuários a manipulam. As questões presentes no questionário exploram algumas das dimensões do *framework*. Em suma, três dimensões foram escolhidas para serem exploradas

a partir das respostas dos participantes: *API Viscosity, Role Expressiveness* e *Work-Step Unit*. Essas dimensões foram escolhidas porque levam em consideração aspectos mais relacionados ao cliente da API, ou seja, os desenvolvedores.

API Viscosity é a dimensão relacionada à dificuldade encarada pelo desenvolvedor na tentativa de alterar um código escrito por determinada API. Já a dimensão *Role Expressiviness* descreve se a API é expressiva ou não, ou seja, quão fácil é entender um código escrito com a determinada API. A última dimensão selecionada, *Work-Step Unit*, está relacionada ao esforço que um programador precisa empregar para cumprir determinada tarefa usando a API.

É interessante destacar que algumas das perguntas presentes no questionário seguiram sugestões do trabalho de Clarke [14], onde o pesquisador descreve questões essenciais que devem ser utilizadas para explorar as dimensões do *framework* de dimensões cognitivas supracitadas.

5.2.2 Resultados

Como dito anteriormente, durante o experimento, foram os próprios participantes quem definiram o caso de teste que implementariam utilizando o SysTest. Então, há a possibilidade de existir implementações para o mesmo caso de teste ou de todos os casos serem diferentes.

Os resultados do experimento estão descritos em quatro partes. Inicialmente, há a caracterização dos participantes. Em seguida serão apresentados todos os números e observações referentes aos testes escritos para o BeeFS. Logo após, o mesmo estudo será feito considerando os testes para o OurGrid. E, a última parte consiste em analisar os resultados do questionário tendo como viés o *framework* de dimensões cognitivas.

Caracterização dos participantes

Foram convidados a participar desse experimento todos os integrantes das equipes de desenvolvimento do BeeFS e do OurGrid. No total participaram 8 programadores, sendo três desenvolvedores do BeeFS e o restante (cinco) do OurGrid. Os participantes possuem experiência com Java variando de 2 a 6 anos, com média igual a 4 anos.

Dentre os desenvolvedores participantes, quatro estão cursando a graduação em Ciência

Número	Caso de Teste
01	Foram definidos dois componentes: o *metadataserver* e o *dataserver*. Cada um é implantado em uma máquina diferente. O teste consiste em verificar o número de *dataservers* conectados ao *metadataserver*
02	Foram definidos três componentes: *Honeycomb*, *Honeybee* e *QueenBee*. Cada um deles foi implantado em uma máquina diferente. O objetivo do teste é verificar se o sumário do sistema é atualizado adequadamente após o tamanho de um arquivo ser alterado com uma operação *truncate*.
03	Foram definidos dois componentes: o *metadataserver* e o *dataserver*. São implantados um *metadataserver* e dois *dataserver*, cada um em uma máquina diferente. O teste consiste em verificar se o espaço de armazenamento disponibilizado pelo *metadataserver* está correto de acordo com conexões e desconexões dos *dataservers*

Tabela 5.1: Descrição dos Casos de Testes para o BeeFS

da Computação na Universidade Federal de Campina Grande. O restante, também 4 programadores, está cursando a pós-graduação em Ciência da Computação do Departamento de Sistemas e Computação (DSC), lotado na mesma universidade. A idade dos participantes varia entre 21 e 25 anos, com média de 23,2 anos.

Testes para o BeeFS

Como mencionado anteriormente, dentre os participantes do experimento, 3 são programadores do projeto BeeFS. Cada um dos 3 desenvolvedores definiu um caso de teste diferente para implementar. A Tabela 5.1 mostra uma breve descrição sobre os testes considerados para o BeeFS.

Como pode ser visto na Tabela 5.1, em 2 testes foram criados 2 componentes do SUT, e, no teste restante foram utilizados 3 componentes. Todos os testes foram implementados através do uso de *Testable Classes* para os componentes do SUT. Em todos os testes implementados, cada componente do SUT executa em uma máquina diferente. Ou seja, em nenhum caso de teste dois componentes do BeeFS executaram na mesma máquina. Com relação ao número de máquinas utilizadas, em dois testes foram usadas 3 máquinas, enquanto

que o outro teste fez uso de 2 máquinas.

Em um primeiro momento, os participantes devem especificar o teste em pseudocódigo. Nesse tipo de experimento, o pseudocódigo representa o que o participante espera que a API que está sendo avaliada disponibilize. Em todos pseudocódigos implementados, os programadores desejam usar abstrações de máquinas para manipular as máquinas do teste. Em dois casos, o programador não especificou como espera que fosse o código para a parte de definição das máquinas, apenas assume que elas estão prontas para serem usadas durante o teste.

O Código 5.10 apresenta o pseudocódigo sugerido por um dos participantes para a escrita do caso de teste 3, o qual está descrito na Tabela 5.1.

Código 5.10: Pseudocódigo para testar caso de teste 03 do BeeFS

```
1  Start up ms m1
2  assert(getAvailableStorage == 0)
3  assert(getOnlineDSs == [])
4
5  start up ds1 m2 (1gb)
6
7  assert(getAvailableStorage == ds1.getAvailableStorage())
8  assert(getOnlineDSs.contains[ds1])
9
10 startup ds2 m3 0.5
11 assert(getAvailableStorage == ds1.getAvailableStorage() + ds2.
       getAvailableStorage())
12 assert(getOnlineDSs.contains[ds2, ds1])
13
14 turn off ds1
15 assert(getAvailableStorage == ds2.getAvailableStorage())
16 assert(getOnlineDSs.contains[ds2])
17
18 turn off ds2
19 assert(getAvailableStorage == 0.0)
20 assert(getOnlineDSs == [])
```

Como pode ser observado no Código 5.10, o programador não especificou como espera que fosse o código para a parte de definição das máquinas. Ele assume que *m1*, *m2* e *m3* são

as máquinas e estão prontas para serem usadas durante o teste. No pseudocódigo referido, o participante faz uso de métodos que na verdade estarão nas *Testable Classes* dos componentes do SUT, tais como: *getAvailableStorage* e *getOnlineDSs*. O SysTest disponibiliza meios para que o programador execute os métodos das *Testable Classes*.

Antes de partir para a escrita das verificações dos testes, o testador precisa montar o cenário de teste. Nessa fase inicial o programador cria as máquinas a serem utilizadas, cria os componentes do SUT e faz a implantação desses nas máquinas. O trecho de código produzido pelo participante para implementar essa etapa é mostrado no Código 5.11.

Código 5.11: Código para montagem do ambiente do teste 03

```
1  Machine msMachine = new DefaultMachine("confFilePathMS");
2  Machine ds1Machine = new DefaultMachine("confFilePathDS2");
3  Machine ds2Machine = new DefaultMachine("confFilePathDS2");
4
5  TestEnvironment test = new TestEnvironment("pathWithTestConfMachines");
6
7  SystemComponent msComponent = new SystemComponent("ms", "/foo/ms.jar", "
       TestableMetadataServer");
8  SystemComponent dsComponent = new SystemComponent("ds", "/foo/ms.jar", "
       TestableDataServer");
9
10 msMachine.deployComponent(msComponent);
11 ds1Machine.deployComponent(dsComponent);
12 ds2Machine.deployComponent(dsComponent);
```

De acordo com as observações feitas, apesar de não tê-lo definido no pseudocódigo, o participante não enfrentou grandes dificuldades para a implementação do Código 5.11. É importante ressaltar que o código dessa etapa pode ser reaproveitado inúmeras vezes. Para isso, basta que existam outros casos de testes que executem nesse mesmo cenário.

O Código 5.12 é o código java produzido através do desenvolvimento do pseudocódigo apresentado no Código 5.10. Mais informações sobre o pseudocódigo, o relacionamento entre o que foi produzido em Código 5.12 e o pseudocódigo no Código 5.10, bem como mais detalhes de implementação do teste e outros exemplos são apresentados no Apêndice D.

Código 5.12: Código produzido para o caso de teste 03 do BeeFS

```
1  //start ms
2  Assert.assertTrue((Boolean)msMachine.executeComponent("ms", "startUp", "
       1234"));
3  //Asserts
4  Assert.assertEquals(0L, (Long) msMachine.executeComponent("ms", "
       getAvailableStorage"));
5  Assert.assertTrue( ( (List<String>) msMachine.executeComponent("ms", "
       getAvailableStorage")).isEmpty());
6  //start ds1
7  int availableDS1 = 1000;
8  Assert.assertTrue((Boolean)ds1Machine.executeComponent("ds", "startUp", "
       msHostname", availableDS1));
9  //Asserts
10 List<String> connectedDSs = new LinkedList<String>();
11 connectedDSs.add("ds1");
12 Assert.assertEquals(availableDS1, (Long)ds1Machine.executeComponent("ds",
       "getAvailableStorage"));
13 Assert.assertEquals(availableDS1, (Long) msMachine.executeComponent("ds",
       "onlineDataServers"));
14 Assert.assertTrue( ( (List<String>) msMachine.executeComponent("ms", "
       getAvailableStorage")).containsAll(connectedDSs)) ;
15 //start ds2
16 int availableDS2 = 500;
17 Assert.assertTrue((Boolean)ds2Machine.executeComponent("ds", "startUp", "
       msHostname", availableDS2));
18 //Asserts
19 connectedDSs.add("ds2");
20 Assert.assertEquals(availableDS2, (Long) ds2Machine.executeComponent("ds"
       , "getAvailableStorage"));
21 Assert.assertEquals(availableDS1 + availableDS2, (Long) msMachine.
       executeComponent("ds", "getAvailableStorage"));
22 Assert.assertTrue( ( (List<String>) msMachine.executeComponent("ms", "
       onlineDataServers")).containsAll(connectedDSs)) ;
23 //turn off ds1
24 Assert.assertTrue((Boolean)ds1Machine.executeComponent("ds", "turnOFF"));
25 //Asserts
26 connectedDSs.remove("ds1");
```

```
27 Assert.assertEquals(availableDS2, (Long) msMachine.executeComponent("ds",
       "getAvailableStorage"));
28 Assert.assertTrue( ( (List<String>) msMachine.executeComponent("ms", "
     onlineDataServers")).containsAll(connectedDSs)) ;
29 //turn off ds2
30 Assert.assertTrue((Boolean)ds2Machine.executeComponent("ds", "turnOFF"));
31 //Asserts
32 connectedDSs.remove("ds2");
33 Assert.assertEquals(0, (Long) msMachine.executeComponent("ds", "
     getAvailableStorage"));
34 Assert.assertTrue( ( (List<String>) msMachine.executeComponent("ms", "
     onlineDataServers")).isEmpty());
```

Apesar de mostrar apenas um dos pseudocódigos, a ideia dos demais participantes do BeeFS foi muito semelhante. Através do pseudcódigo fizeram a descrição do caso de teste. Para isso, utilizaram basicamente os métodos presentes nas *Testable Classes* dos componentes. Em alguns casos, esses métodos retornam objetos de tipos definidos no próprio projeto do BeeFS, os quais são comparados com os objetos esperados através dos *asserts* do JUnit.

O SysTest disponibiliza um meio para o testador executar os métodos, porém não disponibiliza um objeto do tipo *TestableClass* para o programador. Contudo, durante a realização do experimento os participantes não enfrentaram dificuldades para escrever os testes sem utilizar diretamente objetos *TestableClass*, mas sim executar os métodos dessa classe através das abstrações *Machine* e *TestEnvironment*. A alternativa disponível no SysTest é o método *executeComponent*, disponível na interface *Machine* e na classe *TestEnvironment*. Na primeira alternativa, o testador deve passar como parâmetros: o nome do componente a ser executado e já implantado na máquina, o nome e os argumentos do método. Já através da classe *TestEnvironment*, além dos parâmetros anteriores, deve-se informar também a máquina a ser considerada para a execução do componente.

Testes para o OurGrid

No caso do projeto OurGrid, cinco programadores participaram do experimento. Os participantes do OurGrid implementaram casos de testes semelhantes, porém cada um com suas especificidades. Alguns participantes desenvolveram os testes utilizando o SysTest com os

Número	Caso de Teste
01	Implantar um componente *Peer* e o outro *Worker*. O próximo passo é iniciar os componentes. Em seguida, deve-se adicionar o *Worker* no *Peer*, e por último, verificar se os *status* desses componentes estão de acordo com o esperado.
02	Implantar os componentes *Peer*, *Worker* e *Broker*. O próximo passo é iniciar o *Peer*, para em seguida adicionar o *Broker* no *Peer*. Nesse ponto deve-se verificar se o *status* do *Broker* está de acordo com o esperado. Por último, deve-se iniciar o componente *Worker* e verificar seu *status*

Tabela 5.2: Descrição dos Casos de Testes para o OurGrid

scripts já existentes do OurGrid, ou seja, sem a necessidade de implementar as *Testable Classes* dos componentes. Contudo, também existiu participante desenvolvendo e usando as *Testable Classes*.

A Tabela 5.2 apresenta uma breve descrição dos casos de testes implementados para o OurGrid. É interessante notar que apesar de alguns desenvolvedores escreverem casos de testes semelhantes, nenhum deles é igual em sua totalidade, pois utilizaram cenários diferentes (número de máquinas diferente para o mesmo caso de teste), diferentes meios de acesso aos componentes do SUT (usando *scripts* ou *TestableClass*), e, diferentes maneiras de verificar o estado dos componentes (checagem de dados para verificar a ação). Isso acontece devido ao fato de cada programador ter seu próprio perfil e opinião com relação ao caminho mais fácil de realizar determinada tarefa.

O Código 5.13 mostra o pseudocódigo elaborado por um dos participantes para escrever um dos testes. Como pode ser visto nesse exemplo (linhas 1 a 7), o desenvolvedor especificou a etapa de montagem do ambiente. Nesse caso, o programador supõe que os componentes já existem, pois em nenhum momento ele instanciou esses componentes. Também deve ser observado que o programador não especifica como instanciar as máquinas. Ou seja, ele espera que o SysTest disponibilize um meio para a instanciação das máquinas, mas não especificou como seria. Com relação à implantação dos componentes nas máquinas, o SysTest disponibiliza um método com nome de *deployComponent* para essa função.

Código 5.13: Pseudocódigo para caso de teste do OurGrid

```
1  Instanciar m1
2  Instanciar m2
3  Instanciar m3
4
5  Deploy peer em m1
6  Deploy broker em m2
7  Deploy worker em m3
8
9  peer.start()
10 broker.start()
11 worker.start()
12
13 peer.setWorkers([worker])
14 peer.addBroker(broker)
15 broker.setGrid([peer])
16
17 assertEquals(worker.getMasterPeer(), peer)
18 assertTrue(broker.isLogged())
19 assertEquals(broker.getPeer(), peer)
20 assertEquals(peer.getStatus(), {workers=[worker], brokers=[broker]})
```

Observando o Código 5.13, nota-se que a ideia do participante foi similar à dos participantes do BeeFS. Ou seja, os métodos utilizados no pseudocódigo são os que possivelmente estarão no ponto de acesso aos componentes.

O Código 5.14 apresenta o trecho de código para a montagem do cenário necessário ao teste. Ou seja, o trecho que instancia as máquinas e os componentes do SUT para esse teste, bem como faz a implantação dos componentes nas máquinas. Mesmo o trecho de criação dos componentes não sendo lembrado no pseudocódigo, o participante não teve dificuldades durante essa etapa.

Código 5.14: Montagem do cenário para um dos testes do OurGrid

```
1 List<Machine> machines = new LinkedList<Machine>();
2 // Creating machines
3 Machine m1 = new DefaultMachine("conf1");
4 machines.add(m1);
```

```
5  Machine m2 = new DefaultMachine("conf2");
6  machines.add(m2);
7  Machine m3 = new DefaultMachine("conf3");
8  machines.add(m3);
9  //Creating the test case
10 TestEnvironment testCase = new TestEnvironment(machines);
11 //Creating the components
12 SystemComponent peerComponent = new SystemComponent("peer", "ourgrid.jar"
       , "org.ourgrid.system.systest.PeerTestableClass");
13 SystemComponent brokerComponent = new SystemComponent("broker", "ourgrid.
       jar", "org.ourgrid.system.systest.BrokerTestableClass");
14 SystemComponent workerComponent = new SystemComponent("worker", "ourgrid.
       jar", "org.ourgrid.system.systest.WorkerTestableClass");
15 //Deploying components
16 testCase.deployComponent(m1, peerComponent);
17 testCase.deployComponent(m2, brokerComponent);
18 testCase.deployComponent(m3, workerComponent);
```

O autor do Código 5.14 fez uso de *Testable Classes* para escrever esse caso de teste. Para isso, ele precisou desenvolver as seguintes classes: *PeerTestableClass*, *BrokerTestableClass* e *WorkerTestableClass*. Já o Código 5.15 mostra o código produzido a partir do pseudocódigo no Código 5.13.

Código 5.15: Teste implementado para o OurGrid

```
1  //Start the components
2  m1.executeComponent("peer", "start");
3  m2.executeComponent("broker", "start");
4  m3.executeComponent("worker", "start");
5
6  LinkedList<String> workersList = new LinkedList<String>();
7  workersList.add("worker");
8
9  m1.executeComponent("peer", "setWorkers", workersList);
10 m1.executeComponent("peer", "addBroker", "broker");
11 m2.executeComponent("broker", "setGrid", "peer");
12 //Asserts
13 Assert.assertEquals.(m3.executeComponent("worker", "getMasterPeer"), "peer
```

```
    ");
14  Assert.assertTrue((Boolean) m2.executeComponent(    "broker", "isLogged"))
    ;
15  Assert.assertEquals(m2.executeComponent("broker", "getPeer"), "peer");
16
17  PeerCompleteStatus peerStatus = (PeerCompleteStatus) m1.executeComponent(
      "peer", "getStatus");
18  //Asserts
19  Assert.assertEquals(peerStatus.getLocalWorkersInfo().iterator().next().
    getId(), "worker");
20  Assert.assertEquals(peerStatus.getLocalConsumersInfo().iterator().next().
    getConsumerIdentification(), "broker");
```

Como pode ser visto no Código 5.15, assim como nos testes com o BeeFS, o programador faz uso do método *executeComponent* disponibilizado pela interface *Machine* para executar os métodos definidos nas *Testable Classes* dos componentes. Além disso, o participante também faz uso dos *assert* do JUnit para realizar as checagens entre o que se espera e o retorno do método.

Ainda com relação ao Código 5.15, pode ser observado na linha 17 que o retorno do método *getStatus* (definido na *PeerTestableClass*, que é a *TestableClass* do componente *Peer*) é um objeto *PeerCompleteStatus*. Essa classe é definida no projeto do OurGrid. Esse é um exemplo de que com o uso do SysTest com *TestableClass* o programador poderá utilizar objetos de classes do SUT. Com isso, o testador poderá comparar objetos de tipos definidos no projeto do SUT ou utilizar esses objetos para obter os dados necessários ao teste. Um exemplo disso pode ser visto nas linhas 19 a 20 do Código 5.15, onde o participante usou o objeto *peerStatus* para obter os dados necessários a realização dos *asserts*.

Alguns desenvolvedores do OurGrid optaram por não implementar *Testable Classes* para os componentes do mesmo. Em vez delas, eles fizeram uso dos *scripts* já existentes para serem os pontos de acessos aos componentes. O Código 5.16 mostra o código elaborado por um dos participantes para a implementação do caso de teste do número 2 descrito na Tabela 5.2. Esse é o mesmo caso de teste apresentado em pseudocódigo no Código 5.13.

Código 5.16: Teste implementado com scripts existentes para o OurGrid

```
1  Machine machine = new DefaultMachine("testDori.properties");
```

```
2  List<Machine> machines = new ArrayList<Machine>();
3  machines.add(machine);
4  TestEnvironment test = new TestEnvironment(machines);
5
6  //setUp peer
7  machine.executeCommand("mkdir ~/ourgrid/");
8  machine.copyFileToMachine("target/ourgrid-4.2.4-peer.zip", "~/ourgrid/
       ourgrid-4.2.4-peer.zip");
9  machine.executeCommand("unzip ~/ourgrid/ourgrid-4.2.4-peer.zip");
10 machine.copyFileToMachine("peer.properties", "~/ourgrid/peer/peer.
       properties");
11 machine.copyFileToMachine("example.sdf", "~/ourgrid/peer/example.sdf");
12 //setUp worker
13 machine.copyFileToMachine("target/ourgrid-4.2.4-worker.zip", "~/ourgrid/
       ourgrid-4.2.4-worker.zip");
14 machine.executeCommand("unzip ~/ourgrid/ourgrid-4.2.4-worker.zip");
15 machine.copyFileToMachine("worker.properties", "~/ourgrid/worker/worker.
       properties");
16 //setUp broker
17 machine.copyFileToMachine("target/ourgrid-4.2.4-broker.zip", "~/ourgrid/
       ourgrid-4.2.4-broker.zip");
18 machine.executeCommand("unzip ~/ourgrid/ourgrid-4.2.4-broker.zip");
19 machine.copyFileToMachine("~/.broker/broker.properties", "~/.broker/
       broker.properties");
20 //######################
21 //start peer
22 machine.executeCommand("/bash/bin ~/ourgrid/peer/peer start");
23 //add broker to peer
24 machine.executeCommand("/bash/bin ~/ourgrid/peer/peer addbroker broker-
       rodrigo@xmpp.ourgrid.org xmpp-password");
25 //start broker
26 machine.executeCommand("/bash/bin ~/ourgrid/broker/broker start");
27 //wait
28 Thread.sleep(10000);
29 //verify broker status - broker is logged
30 assertTrue(verifyBrokerIsLogged(machine.executeCommand("/bash/bin ~/
       ourgrid/broker/broker status")));
```

```
31 //start worker
32 machine.executeCommand("/bash/bin ~/ourgrid/worker/worker start");
33 //wait
34 Thread.sleep(60000);
35 //verify peer status - worker is idle
36 assertTrue(verifyWorkerIsIdle(machine.executeCommand("/bash/bin ~/ourgrid
     /peer/peer status")));
```

No Código 5.16, das linhas 1 a 19 o desenvolvedor monta o cenário do teste. Como pode ser notado, a montagem do ambiente usando *scripts* difere da montagem com *TestableClass*. Com o uso de *scripts*, não é necessário a instanciação de objetos do tipo *SystemComponent*. O testador escreve o teste automático para fazer o que, na prática, já é feito manualmente ou parcialmente automatizado, ou, no pior caso, não é feito. Portanto, deve-se copiar os arquivos necessários ao teste para a(s) máquina(s) e executar os comandos necessários.

Com relação à execução do caso de teste, o primeiro passo foi iniciar o componente *peer*. Na linha 22 do Código 5.16, o desenvolvedor inicia o *peer* usando o *script* chamado *peer*, o qual foi copiado para máquina durante o *setUp*. Como pode ser notado, o método *executeCommand* da interface *Machine* foi usado para isso. Esse método executa comandos na máquina como se o testador estivesse no *prompt* da mesma.

O retorno do método *executeCommand* é do tipo String. Pois esse retorno é o valor retornado no *prompt* da máquina enquanto executando o comando especificado. Então, para que esse retorno seja utilizado no teste, geralmente é necessário o tratamento do mesmo. O método *verifyBrokerIsLogged* definido no Código 5.17, faz o tratamento do retorno do *executeCommand* para o *assert* da linha 30. Esse método verifica se a String retornada contém **LOGGED**, que é o *status* esperado para o *peer*.

Código 5.17: Métodos para tratar retorno do *executeCommand* para o OurGrid

```
1 private boolean verifyBrokerIsLogged(String command) {
2     return command.contains("LOGGED");
3 }
4
5 private boolean verifyWorkerIsIdle(String command) {
6     return command.contains("IDLE");
7 }
```

O método *verifyWorkerIsIdle*, definido no Código 5.17, faz o tratamento necessário do retorno do comando para o segundo *assert*, na linha 36. Mais informações sobre o pseudocódigo e códigos apresentados, bem como mais detalhes de implementação dos testes realizados e outros exemplos podem ser observados no Apêndice D.

É interessante notar que esse participante definiu, nas linhas 28 e 34 do Código 5.16, um *sleep time*. Isso acontece porque, mesmo as operações disponibilizadas pelo SysTest sendo bloqueantes, o OurGrid é um sistema com operações assíncronas, então, quando o fluxo de execução retorna ao teste não é garantido que a ação tenha sido completamente executada no sistema. Portanto, o uso de um tempo de espera faz com que exista maiores chances de que as operações anteriores à esse tempo tenham concluído suas execuções. Com isso, são evitados falsos positivos, ou seja, é evitado que o teste falhe por conta de no momento da verificação ainda não ter dado tempo das operações anteriores concluirem. Esse problema é recorrente na área de sistemas distribuídos, e, essa alternativa de espera, na prática, é um dos meios utilizados durante o teste de sistemas distribuídos, inclusive nos testes manuais.

No caso do OurGrid, 4 participantes fizeram uso de *scripts* já existentes para serem os pontos de acesso ao sistema. No entanto, um dos participantes definiu e desenvolveu as *Testable Classes* para os componentes. Assim, pode-se verificar que com o uso do SysTest, o testador pode usar o mesmo ambiente de desenvolvimento do SUT até para testar o sistema através de *scripts* que possam já estar desenvolvidos.

Apesar de não apresentar nem detalhar todos os pseudocódigos nem códigos produzidos pelos participantes do experimento, essa seção resumiu bem a ideia dos participantes do OurGrid. A escolha de testes semelhantes ocorreu, segundo os próprios participantes, por conta desses serem os testes de sistema clássicos do OurGrid. Então, no momento de definição do teste a ser escrito, foram de imediato os primeiros testes imaginados. Contudo, esses testes foram escritos de diferentes maneiras, usando diferentes operações e abstrações do SysTest, como também em diferentes cenários.

Análise das Observações e Sugestões

Nessa seção, faremos uma análise dos dados coletados através das anotações feitas pelo condutor do experimento, dos dados obtidos pelo uso do protocolo *Think Aloud* e das respostas do questionário.

Primeiramente, observamos os participantes durante a palestra inicial. Alguns deles, deixaram nítido que não tinham clareza com relação à definição de testes de sistemas. Dois dos participantes mencionaram que relacionavam testes de sistemas com testes que utilizavam objetos mock [46]. Esse fato reforça a ideia de que muitos desenvolvedores não entendem claramente as diferenças entre os difetentes tipos de testes, bem como as vantagens de cada um deles.

Durante a fase de montagem do cenário as máquinas e os componentes são instanciados, a implantação dos componentes é feita, os arquivos necessários ao teste são copiados para as máquinas, algum comando específico é executado nas máquinas, etc. Enfim, essa é a fase do *setUp* do teste. Todos os participantes entenderam o comentário que o cenário montado poderia ser reutilizado por diversos testes. Para isso, bastaria que existissem casos de teste que pudessem executar nesse mesmo cenário. Além disso, os participantes deixaram claro que o esforço vale a pena.

Considerando o desenvolvimento das *Testable Classes* dos componentes, os participantes conseguiram assimilar qual o objetivo da existência dessa classe. Todos os programadores que necessitaram implementá-la definiram apenas os testes necessários ao caso de teste que escreveram. Também é importante mencionar que a implementação dessa classe acontece apenas uma vez, ou seja, com a definição de novos casos de testes que requerem outras operações, bastaria evoluir a *TestableClass* já existente.

A *TestableClass* foi o meio de acesso ao componente do SUT para 50% dos participantes. Desses, 2 desenvolvedores definiram algum método na *TestableClass* que o retorno foi um objeto de um tipo definido no projeto do SUT. Nas *Testable Classes* definidas pelos outros 2 desenvolvedores, os métodos, quando tinham retorno, retornavam tipos primitivos. Contudo, o caso de teste a ser implementado que indica qual será o retorno dos métodos das *Testable Classes*.

O fato de o OurGrid já ter *scripts* desenvolvidos para acessar seus componentes foi primordial para a proporção entre o uso de *TestableClass* e *scripts*. Todos os participantes do BeeFS, o qual não tem *scripts* implementados, optaram pelo uso de *TestableClass*. No caso do OurGrid, apenas um participante definiu as *Testable Classes*. Os demais colocaram em prática a estratégia do reuso, e, utilizaram os *scripts* já prontos. Porém, no momento das verificações, quando em muitos casos é necessário fazer tratamento do retorno dos *scripts*, os

programadores notaram que esse esforço não seria preciso se estivessem utilizando *Testable Classes*.

Durante a escrita dos testes, 06 desenvolvedores apontaram a nomenclatura das abstrações, e, principalmente das operações como uma das qualidades da API. Pois, segundo um deles: "O idioma usado captura facilmente o que o programador está pensando". Outra vantagem apontada, também por 06 dos participantes, foi a existência de uma abstração para representar um componente do SUT. De acordo com outro participante: "As abstrações *Machine* e *SystemComponent* são fáceis de serem compreendidas e utilizadas".

Outro fato que também pode indicar que a API não é complexa, foi que os participantes buscaram utilizar os métodos baseados na assinatura dos mesmos. Ou seja, a documentação da API foi pouco requisitada. Provavelmente por causa disso, um dos participantes cometeu um erro durante a escrita de seu teste. Ele desejava copiar um arquivo da máquina do testador para a do teste, com isso deveria usar o método *copyFileToMachine*. Porém, o testador usou o método *copyFileFromMachine*, o qual copia um arquivo a partir da máquina do teste para a do testador. Outro equívoco cometido por conta da baixa procura à documentação foi a indicação, por parte de um dos participantes, da adição do método *setPathToDeploy* na API, porem o mesmo já existe e está disponível na interface *Machine*.

Através de todos os métodos de coleta de dados utilizados, além de avaliar a usabilidade da API do SysTest, os participantes também puderam sugerir melhorias à mesma. Dois participantes sugeriram a criação de uma abstração que representasse o par <máquina, componente>. Assim, segundo eles, seria possível invocar métodos de uma maneira menos suscetível a erros. A implementação dessa sugestão é viável, inclusive, essa nova abstração poderia ser o retorno do método *deployComponent* da interface *Machine*, pois nesse ponto tanto a máquina como o componente do SUT são conhecidos. Um exemplo de como ficaria a execução de componentes é mostrado no Código 5.18.

Código 5.18: Exemplo de código com a abstração proposta

```
DeployedComponent depC1 = machine.deployComponent(systemComponentObj1)
depC1.invoke("methodName", arg1, ..., argN)
```

Outra sugestão proposta por um dos participantes foi a possibilidade de uma operação *retry(number, condition)* na abstração *TestEnvironment*. Como se sabe, durante a execução de testes de sistemas distribuídos, é possível a ocorrência de falsos-positivos (testes que

podem ser considerados falhos por conta de as execuções anteriores não terem concluído). O método *retry(number, condition)* teria o intuito de tentar evitar os falsos-positivos. Com esse método uma mesma condição seria verificada um número *number* de vezes antes de ser considerada falha. Uma alternativa também seria adicionar o parâmetro *timeBetweenChecks*, o qual representa o tempo de espera entre as verificações. O tempo de espera máximo seria de *number x timeBetweenChecks*. Não há grande complexidade em se adicionar esse método ao SysTest, pois trata-se de um laço de 1 até no máximo *number* vezes, onde a condição será verificada entre intervalos de tempo definidos pelo *timeBetweenChecks*.

Apesar da nomenclatura ter sido apontada como uma das qualidades da API, durante a fase de experimento a abstração de Máquina de Teste foi nomeada de *MachineFake* e dois participantes sugeriram a alteração desse nome. Pois, segundo um deles: "O nome da classe *MachineFake* não traduz o que ela representa". A abstração *MachineFake* foi disponibilizada pelo SysTest para representar uma máquina do teste, porém o testador pode implementar sua própria abstração de máquina e fazê-la implementar a interface *Machine*. Logo, essa classe foi renoameda para *DefaultMachine*.

Ainda com relação a classe *DefaultMachine*, um dos desenvolvedores sugeriu que seu construtor recebesse um objeto *Properties* ao invés do *path* do arquivo de propriedades. Essa sugestão também poderia sem implementada sem dificuldades, pois internamente o que se faz com o arquivo especificado é transformá-lo em um objeto *Properties* da máquina. Assim, é possível adicionar o novo construtor sem eliminar o já existente.

O uso do *framework* JUnit em sua versão mais antiga faz necessário que a classe de teste estenda uma classe chamada *TestCase*. Por conta disso, dois dos participantes, os quais não utilizaram a versão mais recente do JUnit, recomendaram a alteração do nome da classe *TestCase* do SysTest. Portanto, a classe *TestCase* definida durante a execução do experimento foi renomeada para *TestEnvironment*.

O que pôde ser observado ao final do experimento foi o entusiasmo dos participantes com a possibilidade de uso do SysTest para testar suas aplicações. Todos os participantes mencionaram que o SysTest seria muito útil para a fase de testes de sua aplicação. Além disso, o líder da equipe do OurGrid disponibilizou sua equipe para ajudar na evolução do SysTest para que seja realmente utilizado no desenvolvimento do OurGrid.

Resultados do questionário

Os dados coletados através da submissão do questionário aos desenvolvedores serão avaliados nessa seção. Essa análise leva em consideração o *framework* de dimensões cognitivas, pelo qual os aspectos de usabilidade de API serão abordados.

Em relação à dimensão *API Viscosity*, o questionário possuia a seguinte questão:

- Caso fosse necessário alterar o código implementado para o teste, você teria: Muito esforço, esforço razoável ou pouco esforço?

Nesse caso, como pode ser observado na Figura 5.3, 75% dos participantes (6 participantes entre os 8) afirmaram que teriam pouco esforço para alterar o código. Um dos programadores afirmaram que necessitaria de esforço razoável para fazer a modificação e um outro afirmou que empregaria muito esforço. Embora o esforço de uma mudança dependa da própria mudança a ser efetuada, foi observado durante o experimento que os desenvolvedores, por muitas vezes, refatoravam o código do seu teste para uma solução mais simples e que pudesse ser reutilizada mais facilmente adiante. Em alguns casos, o refatoramento do código do teste indicou que a API não se fez um obstáculo para as mudanças.

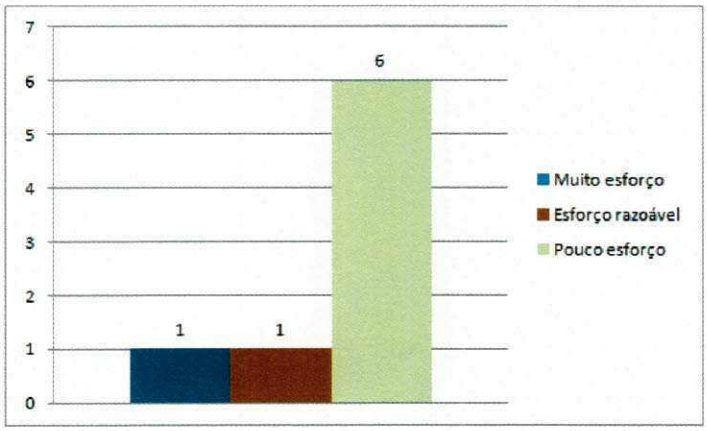

Figura 5.3: Facilidade em alterar o código

Levando em consideração a dimensão *Role Expressiveness*, foram elaboradas das questões:

- Quando você lê o código implementado, quão fácil é entender o que cada parte faz? (Muito fácil, fácil, razoável, difícil e muito difícil)

- Quão fácil é saber qual classe e método, definidos SysTest, usar para a execução de determinada subtarefa? (Muito fácil, fácil, razoável, difícil e muito difícil)

Os Figuras 5.4 e 5.5 apresentam as respostas dos programadores para as questões acima. Em ambos os casos, os resultados indicam que a API é intuitiva e explicativa, considerando que a grande maiorira das respostas variam entre "muito fácil"e "fácil", obtendo-se apenas duas repostas razoável para a primeira e uma para a segunda. A partir do questionário e dos relatos do protocolo *Think Aloud*, identificou-se que 6 dos 8 programadores indicaram que a nomenclatura das abstrações e operações é intuitiva. Esse fato reforça ainda mais os indícios de uma boa usabilidade da API. Além disso, todos os desenvolvedores mostraram-se interessados em usar o SysTest no desenvolvimento de suas aplicações.

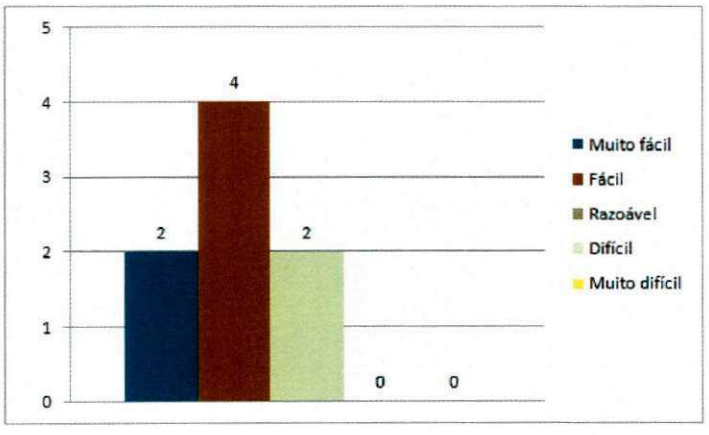

Figura 5.4: Facilidade de mapear o código para a sua função

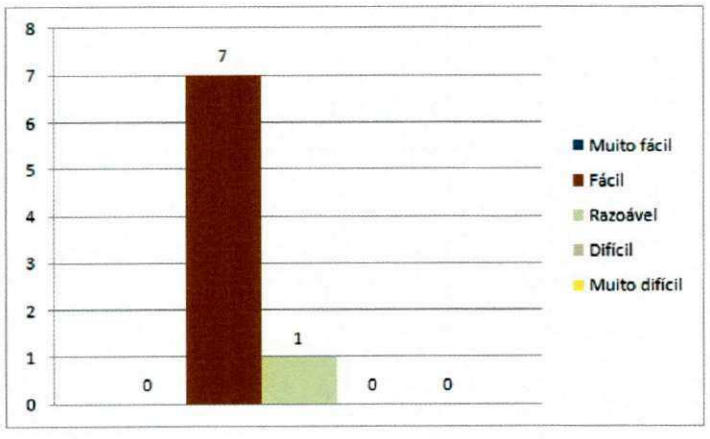

Figura 5.5: Facilidade para mapear entidades do código para determinadas tarefas

Com relação a dimensão *Work-Step Unit* do *framework* de dimensões cognitivas, a seguinte questão foi levantada:

- O que você acha da quantidade de código necessário para implementar o caso de teste? (Grande, justa ou pequena)

Nesse caso, 100% dos participantes afirmaram que a quantidade é "justa"ou "pequena". Isso indica que os programadores acreditam que escrever testes usando o SysTest não é uma tarefa que requer demasiado esforço. Dentre os relatos obtidos com o protocolo *Think Aloud*, dois dos participantes mencionaram o fato de que, com o SysTest, poderiam produzir testes de sistemas através de código bastante semelhante ao código produzido para testes de unidade.

O questionátio também continha um questão relacionada à facilidade em transcrever o pseudocódigo escrito para a implementação concreta usando o SysTest:

- Quão fácil foi traduzir o pseudocódigo para o código do teste?

Embora não esteja relacionada com nenhuma dimensão do *framework* de dimensões cognitivas, esta pergunta foi adicionada baseada na seguinte ideia: se os programadores consideram fácil a tradução do pseudocódigo para a implementação de fato, a API não é um obstáculo na adoção da abordagem de teste. A Figura 5.6 apresenta as respostas relacionadas a essa pergunta.

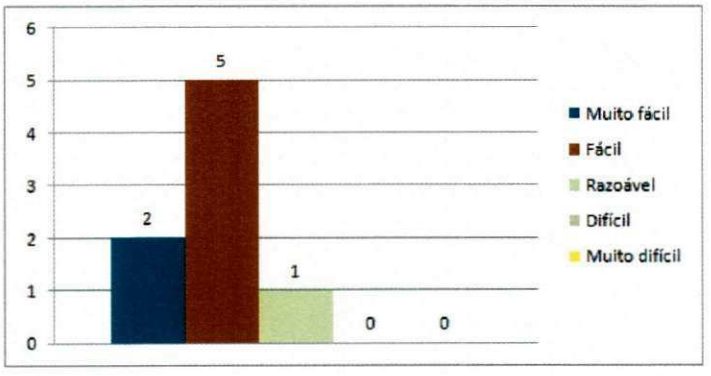

Figura 5.6: Quão fácil é traduzir o pseudocódigo para a implementação de fato

A grande maioria das respostas (7 entre 8) variam entre "muito fácil"e "fácil". Isso indica que a API permitiu que o programador implementasse o que estava descrito em seus

pseudocódigos sem complexidade. O único participante que respondeu "razoável"menciou que esperava chamar os métodos da *Testable Class* em objetos desse tipo, ao contrário do que acontece atualmente, onde esse métodos são chamados a partir da abstração de máquina. Entretanto mesmo assim ficou satisfeito com o SysTest.

O SysTest possui um pré-requisito para o seu uso: o programador precisa deixar sua aplicação testável. A fim de cumprir esse requisito, o desenvolvedor precisa disponibilizar um ponto de acesso para cada componente do SUT que for acionado durante os testes. Então, o questionário também possuiu uma questão relacionada à dificuldade de atender a esse requisito:

- Com relação a deixar o sistema testável, quão fácil foi fazer essa tarefa? (Muito fácil, fácil, razoável, difícil ou muito difícil)

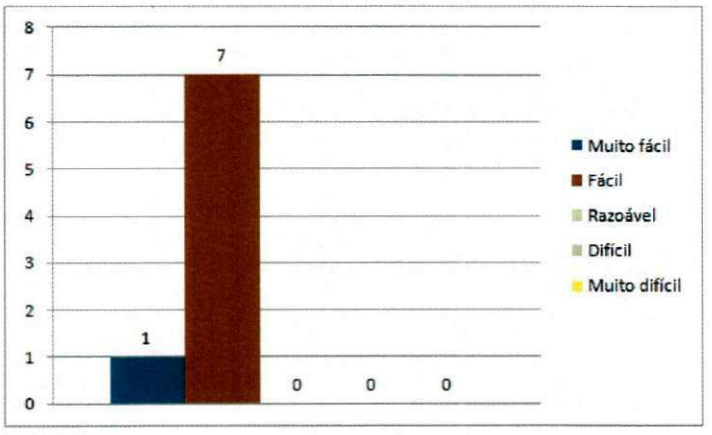

Figura 5.7: Quão fácil é deixar o sistema testável

Como pode ser visto na Figura 5.7, todas as respostas variam entre "muito fácil"e "fácil". Portanto, a existência desse pré-requisito não torna-se uma dificuldade para a viabilidade do SysTest.

Por último, o questionário também teve uma questão sobre a usabilidade da API:

- No geral, qual a sua opinião em relação a usabilidade da API do SysTest? (Muito fácil, fácil, razoável, difícil ou muito difícil)

Essa questão foi colocada no questionário para que o programador que utilizou o SysTest apontando qualidades e possíveis melhorias, pudesse indicar sua opinião sobre a usabilidade do mesmo.

Com relação à usabilidade, 100% dos participantes indicaram que a API do SysTest é fácil de usar. Isso indica que, mesmo observando qualidades e aspectos que podem ser melhorados, a API do SysTest permitiu a os programadores escrever testes de sistema de sua aplicação a partir de sua máquina e utilizando o mesmo ambiente de desenvolvimento usado pelo SUT.

5.2.3 Limitações

Do ponto de vista de pesquisa, é preciso deixar claro que os resultados obtidos durante o experimento apenas dão indícios de que a API é viável e fácil de ser manipulada. Pelo fato de se tratar de um experimento realizado com uma amostra pequena, não é possível garantir que na maioria dos casos os programadores terão facilidade em escrever testes de sistema automáticos de aplicações distribuídas utilizando a API do SysTest.

Logo, não é possível generalizar os resultados obtidos nessa dissertação. De fato, o objetivo desse experimento é relatar a experiência de alguns desenvolvedores ao utilizar a API do SysTest e evidenciar que, de um modo geral, a API desempenhou um bom papel no que diz respeito a sua usabilidade, ou seja, no atendimento aos requisitos dos desenvolvedores.

5.2.4 Conclusões

Nessa seção foi apresentada a avaliação de usabilidade da API do SysTest. A metodologia usada para essa avaliação foi baseada em estudos anteriores com interesses semelhantes. No geral, como visto anteriormente, os desenvolvedores não enfrentaram dificuldades em utilizar a API e todos a classificaram como fácil de ser manipulada.

A avaliação conduzida foi extremamente importante para que fosse possível identificar em casos reais a facilidade que os desenvolvedores possuem ao implementar testes de sistemas de softwares distribuídos com o uso do SysTest. Como também, verificar se a API proposta atende aos requisitos desses desenvolvedores. Além disso, a avaliação teve uma contribuição muito relevante no que diz respeito ao amadurecimento da API do SysTest. Os defeitos e sugestões que se evidenciaram durante a avaliação da API contribuíram para sua evolução. É importante destacar que essa evolução é fruto de sugestões de programadores, com diferentes experiências, utilizando a API do SysTest.

Capítulo 6

Considerações Finais

Nessa dissertação foi apresentada uma abordagem para o desenvolvimento de testes de sistema automáticos de aplicações distribuídas. Um teste de sistema é aquele destinado a testar a aplicação completa e integrada executando em condições similares às quais o software será submetido quando em produção. As características de concorrência e paralelismo de sistemas distribuídos dificultam a realização de testes sobre esse tipo de aplicações, sobretudo a escrita de testes automáticos.

Na tentativa de entender como os projetos de aplicações distribuídas testam seus sistemas, realizamos um *survey*. Baseados nas respostas obtidas, há indícios que a maioria dos projetos realizam testes de sistemas, porém a grande maioria (95% dos respondentes) não consegue automatizá-los completamente. A parcela dos desenvolvedores que conseguem automatizar algum trecho dos testes, utilizam *shell scripts*. Outro fato apontado pelo estudo é que a grande maioria dos desenvolvedores (quase 70% dos participantes) não conhecem qualquer ferramenta que auxilie a execução desse tipo de teste para aplicações distribuídas. No caso dos que conhecem alguma, estes mencionaram ferramentas que apóiam em determinado aspecto os programadores interessados em realizar testes, porém não foram desenvolvidas com esse objetivo.

Então, apresentamos uma abordagem para o desenvolvimento de testes de sistema automáticos de aplicações distribuídas. De acordo com nossa abordagem, o testador deve poder escrever os testes automáticos de sua aplicação utilizando o mesmo ambiente de desenvolvimento usado para a implementação do SUT. De maneira que, através de agentes de teste, o programador pode exercitar o SUT espalhado pela infraestrutura e executar comandos nas

máquinas usadas durante o teste. Com isso, seria possível ao testador manipular e consultar o SUT a partir de sua máquina de maneira automática.

Com o intuito de demonstrar a viabilidade do uso da abordagem proposta, desenvolvemos o SysTest. Esse aplicativo foi desenvolvido em Java e disponibiliza uma API para permitir que os programadores escrevam testes de sistema de aplicações distribuídas em Java. O SysTest oferece abstrações e operações com a finalidade de atender os requisitos da abordagem proposta. A fase de checagem e publicação dos resultados do teste é efetuada pelo *framework* de testes JUnit.

A avaliação do SysTest foi feita sob duas perspectivas:

1. Escrita de casos de testes automáticos, por parte de nossa equipe, para uma aplicação distribuída fictícia implementada por nós;

2. Escrita de casos de testes automáticos de duas aplicações distribuídas reais por parte de seus desenvolvedores.

O desenvolvimento da aplicação fictícia mencionado na avaliação 1, bem como dos casos de testes definidos foram de extrema importância para identificar quais as necessidades que os testadores têm para a realização desse tipo de teste. Através da avaliação 1 conseguimos demonstrar que é possível escrever diferentes casos de testes automáticos para uma aplicação distribuída através do SysTest, bem como a utilização do *framework* JUnit para a checagem e publicação dos resultados do teste. Baseados no uso do SysTest, pudemos identificar em quais aspectos o mesmo poderia evoluir com o inuito de melhorar a legibilidade e simplicidade dos testes automáticos. Porém, a aplicação fictícia não representa um software em produção com suas necessidades reais. Portanto, realizamos a avaliação 2 a fim de avaliar a usabilidade do SysTest no desenvolvimento de testes de softwares distribuídos reais.

O experimento realizado para avaliar a usabilidade da API do SysTest indicou que os programadores não tiveram dificuldades em escrever testes de sistema automáticos utilizando a API, sobretudo pelo fato da linguagem ser a mesma linguagem de programação que eles já conhecem. De fato, através do protocolo *Think Aloud* foi possível observar que a API atendeu, na maioria dos casos, às expectativas dos desenvolvedores na atividade de escrita de testes de sistema automáticos para seu software distribuído.

É preciso deixar claro que, do ponto de vista de pesquisa, os resultados obtidos durante

esses estudos apenas dão indícios de que a API permite a escrita de testes automáticos de aplicações distribuídas e é fácil de ser manipulada. Pois não é possível garantir que na maioria dos casos os programadores terão facilidade em escrever testes de sistema automáticos de aplicações distribuídas utilizando a API do SysTest. Logo, não é possível generalizar os resultados obtidos nessá dissertação.

Também é importante destacar que o SysTest não é uma ferramenta pronta para ser posta em produção. Este é um estudo de caso implementado em Java para a demonstração de viabilidade da abordagem de testes automáticos que propomos.

Como trabalho futuro, poderia ser realizado um estudo considerando a infraestrutura onde os testes são executados. O trabalho apresentado nessa dissertação considera que as máquinas do teste já estão configuradas e prontas para serem utilizadas. Porém, um estudo detalhado sobre como permitir que o testador manipule a infraestutura a partir de sua máquina, iria adicionar mais possibilidades de testes aos programadores. Por exemplo: o programador poderia definir a topologia de rede a ser utilizada pelo teste ou aumentar o tráfego de determinado *link* na rede para verificar como o software se comportaria.

Uma avaliação mais detalhada sobre as sugestões realizadas pelos participantes do experimento como também a implementação das mesmas também devem ser feitas no futuro. Finalmente, colocar o SysTest em produção é o trabalho futuro com maior destaque.

Bibliografia

[1] F. Brasileiro A. Dantas and W. Cirne. Improving automated testing of multi-threaded software. *ICST '08 Proceedings of the 2008 International Conference on Software Testing*, 2008.

[2] R. Gietzel A. Karp I. D. Alderman M. Livny A. Pavlo, P. Couvares and C. Bacon. The nmi build & test laboratory: Continuous integration framework for distributed computing software. *Proceedings of LISA06: Twentieth Systems Administration Conference*, pages 263–273, 2006.

[3] D.P. Anderson. Boinc: A system for public-resource computing and storage. *5th IEEE/ACM International Workshop on Grid Computing*, 4, November 2004.

[4] T. Roscoe A. Bavier L. Peterson M. Wawrzoniak B. Chun, D. Culler and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3), 2003.

[5] J. Stylos B. Ellis and B. Myers. The factory pattern in api design: A usability evaluation. *Proceedings of the 29th internacional coference on Software Engineering*, pages 302–312, 2007.

[6] L. Stoller R. Ricci S. Guruprasad M. Newbold M. Hibler C. Barb B. White, J. Lepreau and A. Joglekar. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review - Proceedings of the 5th symposium on Operating systems design and implementation*, 36(SI):255–270, 2002.

[7] Bamboo. Disponível em: http://www.atlassian.com/software/bamboo/. Acessado em janeiro de 2010.

[8] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2005.

[9] A. Bertolino. Software testing research achievements, challenges, dreams. *International Conference on Software Engineering*, 2007.

[10] J. Blanchette. The little manual of api design. Technical report, Nokia Company, June 2008.

[11] J.W. Silva T.E. Pereira A. Soares C.A. Souza, A.C. Lacerda and F. Brasileiro. Beefs: Um sistema de arquivos distribuído posix barato e eficiente para redes locais. *Simpósio Brasileiro de Redes de Compoutadores 2010 - Tools Session*, Maio 2010.

[12] Capstrano. Disponível https://github.com/capistrano/capistrano/wiki. Acessado em maio de 2011.

[13] S. Clarke. Api usability and cognitive dimensions framework. 2003.

[14] S. Clarke. Measuring api usability. *Doctor Dobbs Journal*, 29(5):1–5, 2004.

[15] S. Clarke. Describing and measuring api usability with cognitive dimensions. *Cognitive Dimensions of Notations 10th Anniversary Workshop*, 2005.

[16] UFCG Universidade Federal de Campina Grande. http://www.ufcg.edu.br.

[17] LSD Laboratório de Sistemas Distribuídos. http://www.lsd.ufcg.edu.br.

[18] JUnit: Framework de Testes de Unidade. http://www.junit.org/. Acessado em maio de 2011.

[19] IDE Eclipse. Disponível em: http://www.eclipse.org/. Acessado em maio de 2011.

[20] ETICS. Disponível http://etics.web.cern.ch/etics/. Acessado em maio de 2011.

[21] Fisher's exact test of independence. Handbook of biological statistics. disponível http://udel.edu/ mcdonald/statfishers.html. Acessado em dezembro de 2009.

[22] Java Remote Method Invocation Distributed Computing for Java. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html. Acessado em maio de 2011.

[23] R. S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6), June 1991.

[24] A. Geraci. Ieee standard computer dictionary: A compilation of ieee standard computer glossaries. *The Institute of Electrical and Electronics Engineers Inc.*, 1991.

[25] J. Dollimore G.F. Coulouris and T. Kindberg. *Distributed Systems - Concepts and Design*. Addison-Wesley, Londres, quarta edição edition, 2006.

[26] T.R.G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[27] M.J. Harrold. Testing: A roadmap. international conference on software engineering. *Proceedings of the Conference on The Future of Software Engineering*, 2000.

[28] D. Guerrero J. Brunet and J. Figueiredo. Structural conformance checking with design tests: An evaluation of usability and scalability. *To appear in Proceedings of the 27th International Conference on Software Maintainance (ICSM 2011)*, 2011.

[29] J.W. Kotrlik J.E. Bartlett and C. Higgins. Organizational research: Determining appropriate sample size for survey research. *Information Technology, Learning, and Performance Journal*, 19(1), 2001.

[30] J. Stylos S.Y.S. Jeong J.K. Beaton, B.A. Myers and Y.C. Xie. Usability evaluation for enterprise soa apis, journal =.

[31] A.E. Treloar K. Pearson, J.A. Harris and M. Wilder. On the theory of contingency. *Journal of the American Statistical Association*, 25(171):320–327, Setembro 1930.

[32] B.A. Kitchenham and S.L. Pfleeger. Principles of survey research part 3: Constructing a survey instrument. *ACM SIGSOFT Software Engineering*, 17(2), Março 2002.

[33] B.A. Kitchenham and S.L. Pfleeger. Principles of survey research part 4: Questionnaire evaluation. *ACM SIGSOFT Software Engineering*, 17(3), Maio 2002.

[34] B.A. Kitchenham and S.L. Pfleeger. Principles of survey research part 5: Populations and samples. *ACM SIGSOFT Software Engineering*, 17(5), Setembro 2002.

[35] C. Lewis. Using the thinking-aloud method in cognitive interface design. *IBM Research Report RC*, 9267(2):17, 1982.

[36] Gerard Mezaros. xunit test patterns refactoring test code. *Addison Wesley*, 2007.

[37] Netbeans. Disponível em: http://netbeans.org/. Acessado em maio de 2011.

[38] A. Lain G. Mecheneau P. Murray P. Goldsack, J. Guijarro and P. Toft. Smartfrog: Configuration and automatic ignition of distributed applications. *HP Openview University Association Conference*, 2003.

[39] R. L. Plackett. Karl pearson and the chi-squared test. *International Statistical Review*, 51(1):59–72, 1983.

[40] Linguagem R. Disponível em http://www.r-project.org.

[41] R. Weber S. Berner and R.K. Keller. Observations and lessons learned from automated testing. *Proceedings of the 27th international Conference on Software Engineering*, 2005.

[42] M.J. Schervish. P values: What they are and what they are not. *The American Statistician*, 50(3):203–206, 1996.

[43] J. Stylos and S. Clarke. Usability implications of requiring parameters in objects' constructor. *ICSE '07: Proceedings of the 29th internacional coference on Software Engineering*, pages 529–239, 2007.

[44] J. Stylos and S. Clarke. Usability implications of requiring parameters in objects' constructor. *Proceedings of the 29th internacional coference on Software Engineering*, pages 529–239, 2007.

[45] J. Stylos and B. A. Myers. The implications of method placement on api learnabiblity. *Proceedings of the 26th ACM SIGSOFT internacional Symposium on Foundations of Software Engineering*, pages 105–112, 2008.

[46] P. Craig T. Mackinnon, S. Freeman. Endo-testing: Unit testing with mock objects. *eXtreme Programming and Flexible Processes in Software Engineering - XP2000*, November 2000.

[47] N. Andrade L.B. Costa A. Andrade R. Novaes W. Cirne, F. Brasileiro and M. Mowbray. Labs of the world, unit!!! *Journal of Grid Computing*, 4(3):224–246, 2006.

Apêndice A

Arquivo de propriedades para a criação de objetos *MachineFake*

Nesse apêndice apresentamos um exemplo de arquivo de propriedades utilizado para a criação de um objeto do tipo *MachineFake*. Além disso definimos as principais propriedades que o testador pode especificar nesse arquivo para o objeto que está sendo criado.

A Figura A.1 mostra o exemplo de um arquivo de propriedades utilizado para a criação de uma *MachineFake*.

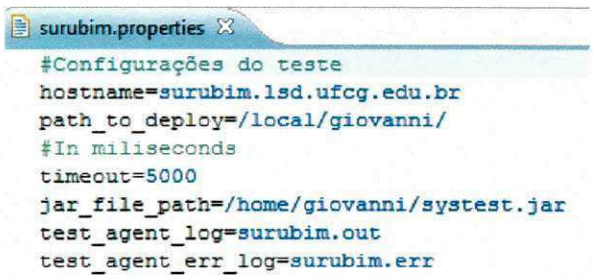

Figura A.1: Exemplo de um arquivo de propriedades para um objeto *MachineFake*

Como pode ser observado na Figura A.1, dentre as propriedades que podem ser especificadas para um objeto *MachineFake*, tem-se:

1. *hostname*: O *hostname* da máquina que será representada pelo objeto que está sendo criado;

2. *path_to_deploy*: O diretório de trabalho na máquina remota, ou seja, todos os coman-

dos que serão excutados na máquina serão feitos a partir deste diretório. Também nesse diretório será armazenada uma cópia do arquivo do SysTest para que o agente de teste seja iniciado posteriormente;

3. *timeout*: O *timeout* das execuções na máquina remota. Portanto, qualquer que seja o comando a ser executado na máquina, se não for encerrado em um espaço de tempo menor que este, a execução é considerada falha;

4. *jar_file_path*: O *path* do arquivo systest.jar na máquina do testador;

5. *test_agent_log*: O nome do arquivo de log do agente do SysTest. Esse arquivo será salvo no diretório de trabalho, e, posteriormente, esse arquivo poderá ser copiado para a máquina do testador para verificação mais detalhada das execuções;

6. *test_agent_err_log*: O nome do arquivo de log para erro do agente do SysTest. Esse arquivo será salvo no diretório de trabalho, e, posteriormente, esse arquivo poderá ser copiado para a máquina do testador para verificação mais detalhada das excessões;

Apêndice B

Questionário para Avaliação de Usabilidade da API do SysTest

Nesse apêndice está descrito o questionário elaborado e aplicado com o objetivo de coletar dados a respeito da opinião dos desenvolvedore de dois projetos de sistemas distribuídos em relação a API do SysTest. Esse questionário foi aplicado individualmente a cada participante através da *web*, onde está disponível no endereço eletrônico https://spreadsheets.google.com/spreadsheet/viewform?formkey=dDh6SjhRYmVwcGU0bGwwWkk2S2VIT1E6MQ.

O questionário foi subdividido em quatro etapas:

1. A caracterização do participante;
2. A caracterização do caso de teste implementado com a API;
3. Esforço empregado para o desenvolvedor cobrir os requisitos para uso da API, ou seja, deixar sua aplicação testável;
4. O *feedback* do participante sobre o uso da API do SysTest.

As perguntas realizadas foram:

Sobre o participante:

1. Quantos anos você tem?
2. Quanto tempo você tem de experiência em Java?

Sobre o caso de teste implementado:

3. Quantas máquinas foram utilizadas?

4. Descreva em poucas palavras o caso de teste que você implementou. Por exemplo, quantos componentes foram criados? Quais componentes executam em quais máquinas?

Sobre a adaptação do SUT para ser testado com o SysTest:

5. Com realação a deixar seu sistema "testável", quão fácil foi fazer essa tarefa? Ou seja, quão fácil foi permitir que sua aplicação fosse testado com a API do SysTest? (Muito fácil, Fácil, Razoável, Difícil ou Muito difícil)

6. Como ponto de acesso aos componentes do seu sistema, você utilizou: (*Testable Class* ou *Shell Script*)

7. Deseja fazer algum comentário sobre essa atividade? Se sim, qual?

Sobre a API do SysTest:

8. Quão fácil foi traduzir o pseudocódigo para o código do teste? (Muito fácil, Fácil, Razoável, Difícil ou Muito difícil)

9. Quando você lê o código implementado, quão fácil é entender o que cada parte faz? (Muito fácil, Fácil, Razoável, Difícil ou Muito difícil)

10. Quão fácil é saber qual classe e método usar para a execução de determinada subtarefa? Por exemplo, realizar um deploy do componente do SUT, ou executar um comando específico na máquina. (Muito fácil, Fácil, Razoável, Difícil ou Muito difícil)

11. O que você acha da quantidade de código necessário para implementar o caso de teste? (Grande, Justa ou Pequena)

12. Caso fosse necessário alterar o código implementado, você teria: (Muito esforçço, Esforço razoável ou Pouco esforço)

13. Aponte defeitos na API que aumentaram a dificuldade enquanto você estava implementado seu caso de teste.

14. Aponte qualidades na API que facilitaram o desenvolvimento do seu caso de teste.

15. Aponte métodos e ou abstrações ausentes na API que aumentaram a dificuldade enquanto você estava implementando o seu caso de teste.

16. No geral, qual sua opinião em relação a usabilidade da API do SysTest? (Muito fácil, Fácil, Razoável, Difícil ou Muito difícil)

17. Deseja adicionar mais alguma observação sobre a API? Se sim, qual?

Apêndice C

Desenvolvimento da aplicação fictícia

Nesse apêndice é apresentado o desenvolvimento da aplicação fictícia utilizada para a avaliação do SysTest. Como aplicação distribuída fictícia foi implementado um aplicativo de mensagens instântaneas obedecendo os requisitos descritos em 3.4.

O aplicativo exemplo possui dois tipos de componentes: Usuário e Servidor. Os Códigos C.1 e C.2 apresentam os códigos relacionados com o componente usuário, bem como os Códigos C.3 e C.4 apresentam os códigos relacionados com o componente servidor. Foi usado Java RMI para a comunicação remota dos objetos da aplicação.

Código C.1: Interface *User*

```
public interface User extends Remote {

    public List<String> getFriends() throws RemoteException;

    public int getNumberOfFriends() throws RemoteException;

    public boolean addFriend(String otherUserID) throws RemoteException;

    public boolean removeFriend(String friendID) throws RemoteException;

    public boolean sendMessage(String message, String destinyFriendID)
        throws RemoteException;

    public boolean disconnectFromServer() throws RemoteException;
```

```
    public boolean connectToServer() throws RemoteException;

    public void unregister() throws RemoteException;

    public String getLastReceivedMessage() throws RemoteException;

    public String getLastSentMessage() throws RemoteException;

    public void receiveMessage(String message, String fromUserID) throws
        RemoteException;
}
```

Código C.2: Classe *UserImpl*

```
public class UserImpl extends UnicastRemoteObject implements User {
    private String name;
    private String password;
    private List<String> friends;
    private String userID;
    private String serverID;
    private String lastReceivedMessage;
    private String lastSentMessage;

    public UserImpl(String name, String password, String serverID, String
        userID) throws RemoteException, MalformedURLException,
        NotBoundException {
        super();
        this.name = name;
        this.password = password;
        friends = new LinkedList<String>();
        this.userID = userID;
        this.serverID = serverID;
        registerOnServer();
        lastReceivedMessage = "";
        lastSentMessage = "";
    }

    private void registerOnServer() throws NotBoundException,
```

```
        MalformedURLException, RemoteException {
    Server server = (Server) Naming.lookup( serverID );
    server.registerUser(this);
}

@Override
public boolean addFriend(String otherUserID) throws RemoteException {
    if (otherUserID!=null && !friends.contains(otherUserID)){
        friends.add(otherUserID);
        User otherUser;
        try {
            otherUser = (User) Naming.lookup( otherUserID );
        } catch (MalformedURLException e) {
            return false;
        } catch (NotBoundException e) {
            return false;
        }
        otherUser.addFriend(userID);
        return true;
    }
    return false;
}

@Override
public int getNumberOfFriends() throws RemoteException {
    return friends.size();
}

@Override
public List<String> getFriends() throws RemoteException {
    return friends;
}

@Override
public boolean removeFriend(String friendID) throws RemoteException {
    if (friendID != null && !"".equals(friendID) && friends.contains(
        friendID)){
```

```
            friends.remove(friendID);
            User otherUser;
            try {
                otherUser = (User) Naming.lookup( friendID );
                otherUser.removeFriend(userID);
                return true;
            } catch (MalformedURLException e) {
                return false;
            } catch (NotBoundException e) {
                return false;
            }
        }
        return false;
    }

    @Override
    public boolean sendMessage(String message, String friendID)
            throws RemoteException {
        User user;
        try {
            user = (User) Naming.lookup( friendID );
            user.receiveMessage(message, userID);
            lastSentMessage = message;
            return true;
        } catch (MalformedURLException e) {
        } catch (NotBoundException e) { }
        return false;
    }

    @Override
    public boolean connectToServer() throws RemoteException {
        Server server;
        try {
            server = (Server) Naming.lookup( serverID );
            server.connectUser(this);
            return true;
        } catch (MalformedURLException e) {
```

```
        } catch (NotBoundException e) { }
        return false;
    }

    @Override
    public boolean disconnectFromServer() throws RemoteException {
        Server server;
        try {
            server = (Server) Naming.lookup( serverID );
            server.disconnectUser(this);
            return true;
        } catch (MalformedURLException e) {
        } catch (NotBoundException e) { }
        return false;
    }

    @Override
    public String getLastReceivedMessage() throws RemoteException {
        return lastReceivedMessage;
    }

    @Override
    public String getLastSentMessage() throws RemoteException {
        return lastSentMessage;
    }

    @Override
    public void unregister() throws RemoteException {
        try{
            Server server = (Server) Naming.lookup( serverID );
            server.unregisterUser(this);
        } catch (MalformedURLException e) {
        } catch (NotBoundException e) { }
    }

    @Override
    public void receiveMessage(String message, String fromUserID) throws
```

```
        RemoteException {
        lastReceivedMessage = message;
    }

    public boolean equals(Object o){
        if (o instanceof UserImpl ){
            UserImpl otherUser = (UserImpl)o;
            if (name.equals(otherUser.getName()) && password.equals(
                otherUser.getPassword())){
                return true;
            }
        }
        return false;
    }

    public String getName() {
        return name;
    }

    public String getPassword() {
        return password;
    }
}
```

Código C.3: Interface *Server*

```
public interface Server extends Remote {

    public boolean registerUser(User user) throws RemoteException;

    public int getNumberOfRegisteredUsers() throws RemoteException;

    public int getNumberOfConnectedUsers() throws RemoteException;

    public boolean connectUser(User user) throws RemoteException;

    public boolean disconnectUser(User user) throws RemoteException;
```

```
    public boolean unregisterUser(User user) throws RemoteException;
}
```

Código C.4: Classe *ServerImpl*

```
public class ServerImpl extends UnicastRemoteObject implements Server {
    private List<User> registeredUsers;
    private List<User> connectedUsers;

    public ServerImpl() throws RemoteException {
        super();
        registeredUsers = new LinkedList<User>();
        connectedUsers = new LinkedList<User>();
    }

    @Override
    public boolean connectUser(User user) throws RemoteException {
        if (user != null){
            connectedUsers.add(user);
            return true;
        }
        return false;
    }

    @Override
    public int getNumberOfConnectedUsers() throws RemoteException {
        return connectedUsers.size();
    }

    @Override
    public int getNumberOfRegisteredUsers() throws RemoteException {
        return registeredUsers.size();
    }

    @Override
    public boolean registerUser(User user) throws RemoteException {
        if (user != null){
            registeredUsers.add(user);
```

```
            return true;
        }
        return false;
    }

    @Override
    public boolean disconnectUser(User user) throws RemoteException {
        if (user != null && connectedUsers.contains(user)){
            connectedUsers.remove(user);
            return true;
        }
        return false;
    }

    @Override
    public boolean unregisterUser(User user) throws RemoteException {
        if (user != null && registeredUsers.contains(user)){
            registeredUsers.remove(user);
            return true;
        }
        return false;
    }
}
```

Para permitir que esse aplicativo seja testado usando o SysTest, foram criadas as *Testable Classes* do usuário e servidor. A *Testable Class* de um componente não é uma classe estática, ou seja, ela pode evoluir de acordo com as necessidades do testador. Ao longo desse trabalho, as *Testable Classes* evoluiram de acordo com as funcionalidades que os casos de testes exploravam. Os Códigos C.5 e C.6 mostram o código final para as *Testable Classes* do usuário e servidor respectivamente.

Código C.5: *Testable Class* do componente usuário

```
public class TestableUserClass extends TestableImpl{
    User user;
    String userID;

    public TestableUserClass() throws RemoteException {
```

```
        super();
    }

    public String registerUser(String name, String password, String
        serverID) throws RemoteException, MalformedURLException,
        UnknownHostException, NotBoundException{
        userID = "rmi://" + InetAddress.getLocalHost().getHostName() + "
            :1099/" + name;
        user = new UserImpl(name, password, serverID, userID);
        Naming.rebind(userID, user);
        return userID;
    }

    public boolean connectToServer() throws MalformedURLException,
        RemoteException, NotBoundException{
        return user.connectToServer();
    }

    public int getNumberOfFriends() throws RemoteException{
        return user.getNumberOfFriends();
    }

    public boolean addFriend(String otherUserID) throws RemoteException,
        MalformedURLException, NotBoundException{
        return user.addFriend(otherUserID);
    }

    public void stopUser() throws RemoteException, MalformedURLException,
        NotBoundException {
        Naming.unbind(userID);
    }

    public boolean disconnect() throws RemoteException{
        return user.disconnectFromServer();
    }

    public void unregister() throws RemoteException{
```

```
        user.unregister();
    }

    public String getLastReceivedMessage() throws RemoteException{
        return user.getLastReceivedMessage();
    }

    public String getLastSentMessage() throws RemoteException{
        return user.getLastSentMessage();
    }

    public boolean sendMessage(String message, String userID) throws
        RemoteException{
        return user.sendMessage(message, userID);
    }

    public boolean removeFriend(String friendID) throws RemoteException {
        return user.removeFriend(friendID);
    }
}
```

Código C.6: *Testable Class* do componente servidor

```
public class TestableServerClass extends TestableImpl {
    Server server;
    String serverID;

    public TestableServerClass() throws RemoteException {
        super();
    }

    public String startServer(String serverName) throws RemoteException,
        MalformedURLException, UnknownHostException{
        server = new ServerImpl();
        serverID = "rmi://" + InetAddress.getLocalHost().getHostName() + "
            :1099/" + serverName;
        Naming.rebind(serverID, server);
        return serverID;
```

```
    }

    public int getNumberOfRegisteredUsers() throws RemoteException{
        return server.getNumberOfRegisteredUsers();
    }

    public int getNumberOfConnectedUsers() throws RemoteException{
        return server.getNumberOfConnectedUsers();
    }

    public void stopServer() throws RemoteException, MalformedURLException
        , NotBoundException{
        Naming.unbind(serverID);
    }
}
```

Apêndice D

Detalhes de implementação dos testes produzidos durante a avaliação de usabilidade do SysTest

Nesse apêndice são apresentados detalhes sobre os códigos visto durante a seção de avaliação de usabilidade e que foram produzidos pelos participantes do experimento.

D.1 Testes para o BeeFS

O Código D.1 foi apresentado no Capítulo 5 e é o pseudocódigo sugerido por um dos participantes para a escrita de um caso de teste do BeeFS. Nesse pseudocódigo, o programador não especificou como espera que fosse o código para a parte de definição das máquinas. Ele assume que *m1*, *m2* e *m3* são as máquinas e estão prontas para serem usadas durante o teste.

Código D.1: Pseudocódigo para testar caso de teste 03 do BeeFS

```
1 Start up ms m1
2 assert(getAvailableStorage == 0)
3 assert(getOnlineDSs == [])
4
5 start up ds1 m2 (1gb)
6
7 assert(getAvailableStorage == ds1.getAvailableStorage())
8 assert(getOnlineDSs.contains[ds1])
```

```
9
10 startup ds2 m3 0.5
11 assert(getAvailableStorage == ds1.getAvailableStorage() + ds2.
       getAvailableStorage())
12 assert(getOnlineDSs.contains[ds2, ds1])
13
14 turn off ds1
15 assert(getAvailableStorage == ds2.getAvailableStorage())
16 assert(getOnlineDSs.contains[ds2])
17
18 turn off ds2
19 assert(getAvailableStorage == 0.0)
20 assert(getOnlineDSs == [])
```

Nesse tipo de experimento, o pseudocódigo representa o que o participante espera que a API que está sendo avaliada disponibilize. Na tentativa de esclarecer um pouco mais o pseudocódigo mostrado no Código 5.10, pode-se notar que:

- na linha 1, o participante espera poder iniciar o componente *metadataserver*, chamado de *ms*, na máquina *m1*;
- nas linhas 2 e 3, deseja ter como verificar se as características iniciais estão de acordo com o esperado;
- na linha 5, o programador quer iniciar o componente *dataserver*, chamado de *ds1*, na máquina *m2* passando 1Gb como argumento;
- das linhas 7 a 8, o desenvolvedor espera ter como verificar se os dados foram atualizados. Pois nesse ponto o *ms* deve ter o *ds1 online*, como também o *avaiableStorage* do *ms* deve ser igual ao *avaiableStorage* de *ds1*;
- entre as linhas 10 e 12, o programador espera poder iniciar o *ds2* em *m3* passando 0.5Gb como argumento, e, verificar se os dados foram atualizados no componente *ms*;
- entre as linhas 14 e 16, o participante deseja poder desligar *ds1*, e, verificar se o componente *ms* atualiza os dados;

- por último, da linha 18 a 20, ele quer desligar o *ds2*, e, novamente verificar se o componente *ms* atualiza os dados.

Antes de partir para a escrita do testes propriamente dito, o testador precisa montar o cenário de teste. Nessa fase inicial o programador cria as máquinas a serem utilizadas, cria os componentes do SUT e faz a implantação desses nas máquinas. O trecho de código produzido pelo participante para implementar essa etapa é mostrado no Código D.2.

Na linha 5 do Código D.2, encontra-se um erro. A classe *TestCase* possui dois construtores. Em um deles recebe como parâmetro uma lista de máquinas que participarão do teste. No outro construtor, recebe o *path* de um arquivo que contém os *paths* para os arquivos de propriedades das máquinas. Nesse caso, o próprio construtor instancia as máquinas. No Código D.2, o programador instanciou as máquinas (linhas 1 a 3) e criou o objeto *TesteCase* passando o arquivo como parâmetro, quando na verdade deveria passar as máquinas já instancidas. Esse erro poderia ter sido evitado através de uma breve leitura na documentação do SysTest, a qual, de acordo com as observações do condutor do experimento, foi pouco utilizada pelos participantes.

De acordo com as observações feitas, apesar do erro cometido por esse participante e mesmo não especificando essa etapa no pseudocódigo, o participante não enfrentou grandes dificuldades para a realização dessa tarefa. É importante ressaltar que o código dessa etapa pode ser reaproveitado inúmeras vezes. Para isso, basta que existam outros casos de testes que executem nesse mesmo cenário.

Código D.2: Código para montagem do ambiente do teste 03

```
1 Machine msMachine = new DefaultMachine("confFilePathMS");
2 Machine ds1Machine = new DefaultMachine("confFilePathDS2");
3 Machine ds2Machine = new DefaultMachine("confFilePathDS2");
4
5 TestEnvironment test = new TestEnvironment("pathWithTestConfMachines");
6
7 SystemComponent msComponent = new SystemComponent("ms", "/foo/ms.jar", "
    TestableMetadataServer");
8 SystemComponent dsComponent = new SystemComponent("ds", "/foo/ms.jar", "
    TestableDataServer");
9
```

```
10 msMachine.deployComponent(msComponent);
11 ds1Machine.deployComponent(dsComponent);
12 ds2Machine.deployComponent(dsComponent);
```

O Código D.3 é o código java produzido através do desenvolvimento do pseudocódigo apresentado no Código D.1. O relacionamento entre o que foi produzido em Código D.3 e o pseudocódigo em Código D.1 é feito a fim de verificar se o SysTest disponibiliza o que o testador necessita.

A linha 2 no Código D.3 inicia o componente *metadataserver* na máquina *msMachine* seguindo as instruções do método *startUp* presente na *Testable Class* do componente *metadataserver*. Essa linha corresponde à linha 1 no Código 5.10. Porém, além de iniciar o *metadataserver*, o desenvolvedor do SUT fez com que o método *startUp* retornasse um valor booleano para indicar se houve problemas ou não durante a inicialização. Portanto, na linha 2 do Código D.3, o testador inicia o *metadataserver* e já realiza o primeiro *assert* do teste.

As linhas 4 a 5 do Código D.3 correspondem as verificações iniciais especificadas nas linhas 2 a 3 do o Código D.1. Já ao longo das linhas 6 a 21 do Código D.3, trata-se da inicialização dos *dataservers* nas máquinas *ds1Machine* e *ds2Machine*, bem como as verificações se os componentes estão inicializando corretamente e os dados estão sendo atualizados no componente *metadataserver*. Esse trecho corresponde às linhas 5 a 12 do peseudocódigo apresentado no Código D.1. Por fim, nas linhas 23 a 24, ocorre o desligamento dos *dataservers* e as checagens se o *metadataserver* atualiza os dados corretamente. Essa etapa final corresponde as linhas 14 a 20 do pseudocódigo supracitado.

Código D.3: Código produzido para o caso de teste 03 do BeeFS

```
1 //start ms
2 Assert.assertTrue((Boolean)msMachine.executeComponent("ms", "startUp", "
    1234"));
3 //Asserts
4 Assert.assertEquals(0L, (Long) msMachine.executeComponent("ms", "
    getAvailableStorage"));
5 Assert.assertTrue( ( (List<String>) msMachine.executeComponent("ms", "
    getAvailableStorage")).isEmpty());
6 //start ds1
```

```
7  int availableDS1 = 1000;
8  Assert.assertTrue((Boolean)ds1Machine.executeComponent("ds", "startUp", "
       msHostname", availableDS1));
9  //Asserts
10 List<String> connectedDSs = new LinkedList<String>();
11 connectedDSs.add("ds1");
12 Assert.assertEquals(availableDS1, (Long)ds1Machine.executeComponent("ds",
        "getAvailableStorage"));
13 Assert.assertEquals(availableDS1, (Long) msMachine.executeComponent("ds",
        "onlineDataServers"));
14 Assert.assertTrue( ( (List<String>) msMachine.executeComponent("ms", "
       getAvailableStorage")).containsAll(connectedDSs)) ;
15 //start ds2
16 int availableDS2 = 500;
17 Assert.assertTrue((Boolean)ds2Machine.executeComponent("ds", "startUp", "
       msHostname", availableDS2));
18 //Asserts
19 connectedDSs.add("ds2");
20 Assert.assertEquals(availableDS2, (Long) ds2Machine.executeComponent("ds"
       , "getAvailableStorage"));
21 Assert.assertEquals(availableDS1 + availableDS2, (Long) msMachine.
       executeComponent("ds", "getAvailableStorage"));
22 Assert.assertTrue( ( (List<String>) msMachine.executeComponent("ms", "
       onlineDataServers")).containsAll(connectedDSs)) ;
23 //turn off ds1
24 Assert.assertTrue((Boolean)ds1Machine.executeComponent("ds", "turnOFF"));
25 //Asserts
26 connectedDSs.remove("ds1");
27 Assert.assertEquals(availableDS2, (Long) msMachine.executeComponent("ds",
        "getAvailableStorage"));
28 Assert.assertTrue( ( (List<String>) msMachine.executeComponent("ms", "
       onlineDataServers")).containsAll(connectedDSs)) ;
29 //turn off ds2
30 Assert.assertTrue((Boolean)ds2Machine.executeComponent("ds", "turnOFF"));
31 //Asserts
32 connectedDSs.remove("ds2");
```

```
33 Assert.assertEquals(0, (Long) msMachine.executeComponent("ds", "
       getAvailableStorage"));
34 Assert.assertTrue( ( (List<String>) msMachine.executeComponent("ms", "
       onlineDataServers")).isEmpty());
```

A ideia dos demais participantes do BeeFS foi muito semelhante. Através do pseudcódigo fizeram a descrição do caso de teste. Para isso, utilizaram basicamente os métodos presentes nas *Testable Classes* dos componentes. O SysTest disponibiliza um meio para o testador executar os métodos das *Testable Classes*. Em alguns casos, esses métodos retornam objetos de tipos definidos no próprio projeto do BeeFS, os quais são comparados com os objetos esperados através dos *asserts* do JUnit.

D.2 Testes para o OurGrid

O Código D.4 mostra o pseudocódigo elaborado por um dos participantes para escrever um dos testes. Como pode ser visto nesse exemplo, o desenvolvedor especificou a etapa de montagem do ambiente. Da linha 1 a 3 ele instancia as máquinas, e em seguida (linhas 5 a 7) faz a implantação dos componentes nas mesmas. Nesse caso, o programador supõe que os componentes já existem, pois em nenhum momento ele instanciou esses componentes. Também deve ser observado que o programador não especifica como instanciar as máquinas. Ou seja, ele espera que o SysTest disponibilize um meio para a instanciação das máquinas, mas não especificou como seria. Com relação à implantação dos componentes nas máquinas, o SysTest disponibiliza um método com nome de *deployComponent* para essa função.

Ainda considerando o pseudocódigo no Código D.4, após o *deploy* dos componentes, o participante deseja iniciar os componentes (linhas 9 a 11). Em seguida, nas linhas 13 a 15, são realizadas as ações a serem realizadas durante o teste. Por último, são realizados os *asserts* com as verificações desejadas (linha 17 a 20).

Código D.4: Pseudocódigo para caso de teste do OurGrid

```
1 Instanciar m1
2 Instanciar m2
3 Instanciar m3
4
5 Deploy peer em m1
```

```
6  Deploy broker em m2
7  Deploy worker em m3
8
9  peer.start()
10 broker.start()
11 worker.start()
12
13 peer.setWorkers([worker])
14 peer.addBroker(broker)
15 broker.setGrid([peer])
16
17 assertEquals(worker.getMasterPeer(), peer)
18 assertTrue(broker.isLogged())
19 assertEquals(broker.getPeer(), peer)
20 assertEquals(peer.getStatus(), {workers=[worker], brokers=[broker]})
```

O Código D.5 apresenta o código para a montagem do cenário necessário ao teste. Ou seja, o trecho que instancia as máquinas e os componentes do SUT para esse teste, bem como faz o *deploy* dos componentes nas máquinas. Mesmo o trecho de criação dos componentes não sendo lembrado no pseudocódigo, o participante não teve dificuldades durante essa etapa.

Código D.5: Montagem do cenário para um dos testes do OurGrid

```
1  List<Machine> machines = new LinkedList<Machine>();
2  //Creating machines
3  Machine m1 = new DefaultMachine("conf1");
4  machines.add(m1);
5  Machine m2 = new DefaultMachine("conf2");
6  machines.add(m2);
7  Machine m3 = new DefaultMachine("conf3");
8  machines.add(m3);
9  //Creating the test case
10 TestEnvironment testCase = new TestEnvironment(machines);
11 //Creating the components
12 SystemComponent peerComponent = new SystemComponent("peer", "ourgrid.jar"
       , "org.ourgrid.system.systest.PeerTestableClass");
13 SystemComponent brokerComponent = new SystemComponent("broker", "ourgrid.
```

```
    jar", "org.ourgrid.system.systest.BrokerTestableClass");
14  SystemComponent workerComponent = new SystemComponent("worker", "ourgrid.
    jar", "org.ourgrid.system.systest.WorkerTestableClass");
15  //Deploying componènts
16  testCase.deployComponent(m1, peerComponent);
17  testCase.deployComponent(m2, brokerComponent);
18  testCase.deployComponent(m3, workerComponent);
```

Pode ser observado no Código D.5, que esse participante fez uso de *Testable Classes* para escrever esse caso de teste. Para isso, ele precisou desenvolver as seguintes classes: *PeerTestableClass*, *BrokerTestableClass* e *WorkerTestableClass*. É interessante destacar que esse trecho poderá ser reutilizado para todos os casos de testes que possam executar nesse cenário.

Já o Código D.6 mostra o código produzido a partir do pseudocódigo no Código D.4. Como pode ser visto, assim como nos testes com o BeeFS, o programador faz uso do método *executeComponent* disponibilizado pela interface *Machine* para executar os métodos definidos nas *Testable Classes* dos componentes. Além disso, o participante também faz uso dos *assert* do JUnit para realizar as checagens entre o que se espera e o retorno do método.

Código D.6: Teste implementado para o OurGrid

```
1  //Start the components
2  m1.executeComponent("peer", "start");
3  m2.executeComponent("broker", "start");
4  m3.executeComponent("worker", "start");
5
6  LinkedList<String> workersList = new LinkedList<String>();
7  workersList.add("worker");
8
9  m1.executeComponent("peer", "setWorkers", workersList);
10 m1.executeComponent("peer", "addBroker", "broker");
11 m2.executeComponent("broker", "setGrid", "peer");
12 //Asserts
13 Assert.assertEquals(m3.executeComponent("worker", "getMasterPeer"), "peer
   ");
14 Assert.assertTrue((Boolean) m2.executeComponent(   "broker", "isLogged"))
   ;
```

```
15 Assert.assertEquals(m2.executeComponent("broker", "getPeer"), "peer");
16
17 PeerCompleteStatus peerStatus = (PeerCompleteStatus) m1.executeComponent(
       "peer", "getStatus");
18 //Asserts
19 Assert.assertEquals(peerStatus.getLocalWorkersInfo().iterator().next().
       getId(), "worker");
20 Assert.assertEquals(peerStatus.getLocalConsumersInfo().iterator().next().
       getConsumerIdentification(), "broker");
```

Ainda com relação ao Código D.6, pode ser observado na linha 17 que o retorno do método *getStatus* (definido na *PeerTestableClass*, que é a *Testable Class* do componente *Peer*) é um objeto *PeerCompleteStatus*. Essa classe é definida no projeto do OurGrid. Esse é um exemplo de que com o uso do SysTest com *Testable Class* o programador poderá utilizar objetos de classes do SUT. Com isso, o testador poderá comparar objetos de tipos definidos no projeto do SUT ou utilizar esses objetos para obter os dados necessários ao teste. Um exemplo disso pode ser visto nas linhas 19 a 20 do Código D.6. Nesse trecho do código, o participante usou o objeto *peerStatus* para obter os dados necessários a realização dos *asserts*.

Alguns desenvolvedores do OurGrid optaram por não implementar *Testable Classes* para os componentes do mesmo. Em vez delas, eles fizeram uso dos *scripts* já existentes para serem os pontos de acessos aos componentes. O Código 5.16 mostra o código elaborado por um dos participantes para a implementação do caso de teste apresentado em pseudocódigo no Código D.4.

Código D.7: Teste implementado com scripts existentes para o OurGrid

```
1 Machine machine = new DefaultMachine("testDori.properties");
2 List<Machine> machines = new ArrayList<Machine>();
3 machines.add(machine);
4 TestEnvironment test = new TestEnvironment(machines);
5
6 //setUp peer
7 machine.executeCommand("mkdir ~/ourgrid/");
8 machine.copyFileToMachine("target/ourgrid-4.2.4-peer.zip", "~/ourgrid/
      ourgrid-4.2.4-peer.zip");
```

```
9  machine.executeCommand("unzip ~/ourgrid/ourgrid-4.2.4-peer.zip");
10 machine.copyFileToMachine("peer.properties", "~/ourgrid/peer/peer.
       properties");
11 machine.copyFileToMachine("example.sdf", "~/ourgrid/peer/example.sdf");
12 //setUp worker
13 machine.copyFileToMachine("target/ourgrid-4.2.4-worker.zip", "~/ourgrid/
       ourgrid-4.2.4-worker.zip");
14 machine.executeCommand("unzip ~/ourgrid/ourgrid-4.2.4-worker.zip");
15 machine.copyFileToMachine("worker.properties", "~/ourgrid/worker/worker.
       properties");
16 //setUp broker
17 machine.copyFileToMachine("target/ourgrid-4.2.4-broker.zip", "~/ourgrid/
       ourgrid-4.2.4-broker.zip");
18 machine.executeCommand("unzip ~/ourgrid/ourgrid-4.2.4-broker.zip");
19 machine.copyFileToMachine("~/.broker/broker.properties", "~/.broker/
       broker.properties");
20 //######################
21 //start peer
22 machine.executeCommand("/bash/bin ~/ourgrid/peer/peer start");
23 //add broker to peer
24 machine.executeCommand("/bash/bin ~/ourgrid/peer/peer addbroker broker-
       rodrigo@xmpp.ourgrid.org xmpp-password");
25 //start broker
26 machine.executeCommand("/bash/bin ~/ourgrid/broker/broker start");
27 //wait
28 Thread.sleep(10000);
29 //verify broker status - broker is logged
30 assertTrue(verifyBrokerIsLogged(machine.executeCommand("/bash/bin ~/
       ourgrid/broker/broker status")));
31 //start worker
32 machine.executeCommand("/bash/bin ~/ourgrid/worker/worker start");
33 //wait
34 Thread.sleep(60000);
35 //verify peer status - worker is idle
36 assertTrue(verifyWorkerIsIdle(machine.executeCommand("/bash/bin ~/ourgrid
       /peer/peer status")));
```

No Código D.7, das linhas 1 a 19 o desenvolvedor monta o cenário do teste. Como pode

ser notado, a montagem do ambiente usando *scripts* difere da montagem com *Testable Class*. Com o uso de *scripts*, não é necessário a instanciação de objetos do tipo *SystemComponent*. O testador escreve o teste automático para fazer o que, na prática, já é feito manualmente ou parcialmente automatizado, ou, no pior caso, não é feito. Portanto, deve-se copiar os arquivos necessários ao teste para a(s) máquina(s) e executar os comandos necessários.

Entre as linhas 6 e 15 do Código D.7 é realizado o *setUp* do componente *peer*. Na linha 7, o testador cria o diretório *"~ourgrid"* na máquina *machine* através do comando *mkdir* executado usando o método *excuteCommand* da interface *Machine*. Em seguida, na linha 8, o programador copia o arquivo *"target/ourgrid-4.2.4-peer.zip"* para *machine* usando o método *copyFileToMachine*. Logo após, na linha 9, descompacta o arquivo copiado para *machine* novamente usando o método *executeCommand*. Para encerrar o *setUp* do *peer*, nas linhas 10 e 11 os arquivos necessários são copiados para *machine* através do método *copyFileToMachine*.

Após deixar a *machine* pronta para o *peer* executar, o desenvolvedor fez o *setUp* dos outros componentes. Entre as linhas 12 e 15 do Código D.7, foi feito o *setUp* do *worker*. Já o do componente *broker* foi realizado entre as linhas 16 e 19. O cenário montado para o caso de teste envolve apenas uma máquina, ou seja, todos os componentes executam na mesma máquina.

Até a linha 20 do Código D.7 apenas foi feito o *setUp* dos componentes. É importante frisar que esse código pode ser reutilizado inúmeras vezes, basta que existam mais casos de testes do OurGrid que possam executar nesse mesmo cenário.

Com relação à execução do caso de teste, o primeiro passo foi iniciar o componente *peer*. Na linha 22 do Código D.7, o desenvolvedor inicia o *peer* usando o *script* chamado *peer*, o qual foi copiado para máquina durante o *setUp*. Como pode ser notado, o método *executeCommand* da interface *Machine* foi usado para isso. Esse método executa comandos na máquina como se o testador estivesse no *prompt* da mesma.

Nas linhas 23 e 28, o particpante faz as configurações definidas pelo caso de teste, novamente usando o método *executeCommand*. O primeiro *assert* do teste é realizado na linha 30. Para isso, o programador executou o comando de verificação do *status* do *broker*. O retorno do método *executeCommand* é do tipo String. Pois esse retorno é o valor retornado no *prompt* da máquina enquanto executando o comando especificado. Então, para que esse

retorno seja utilizado no teste, é necessário o tratamento do mesmo. O método *verifyBrokerIsLogged* definido no Código D.8, faz o tratamento do retorno do *executeCommand* para o *assert* da linha 30. Esse método verifica se a String retornada contém **LOGGED**, que é o *status* esperado para o *peer*.

Código D.8: Métodos para tratar retorno do *executeCommand* par o OurGrid

```
1 private boolean verifyBrokerIsLogged(String command) {
2     return command.contains("LOGGED");
3 }
4
5 private boolean verifyWorkerIsIdle(String command) {
6     return command.contains("IDLE");
7 }
```

Após o primeiro *assert*, na linha 30, o *worker* é iniciado. Na linha 36, é feito um outro *assert* para verificar o *status* no *peer*. O método *verifyWorkerIsIdle*, definido no Código D.8, faz o tratamento necessário do retorno do comando para o segundo *assert*.