Pós-Graduação em Ciência da Computação

"FlexLoadGenerator - Um Framework para Apoiar o Desenvolvimento de Ferramentas Voltadas a Estudos de Avaliação de Desempenho e Dependabilidade"

Por

Débora Stefani Lima de Souza

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, AGOSTO/2013

Universidade Federal de Pernambuco
Centro de Informática
Pós-graduação em Ciência da Computação

Débora Stefani Lima de Souza

"FlexLoadGenerator - Um Framework para Apoiar o Desenvolvimento de Ferramentas Voltadas a Estudos de Avaliação de Desempenho e Dependabilidade"

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Prof. Dr. Paulo Romero Martins Maciel*

RECIFE, AGOSTO/2013

A Deus e a minha família que sempre esteve ao meu lado
em todos os momentos. . . .

Agradecimentos

Meus sinceros agradecimentos ao meu orientador Prof. Dr. Paulo Romero Martins Maciel pelo apoio, disponibilidade e acima de tudo pela excelente orientação prestada sem a qual este trabalho não teria sido possível.

Agradeço também aos professores Nelson Rosa e Gabriel Alves por aceitarem o convite para compor a banca examinadora e pelas contribuições oferecidas.

Meu agradecimento especial a Rubens Matos que muito me ajudou ao longo dessa jornada. Quero agradecer também aos colegas do grupo *MoDCS*, dentre eles: Jean Araújo, Vandi Alves, Rafael Roque, Jamilson Dantas, Pedro Dias e Julian Araújo. Gostaria de agradecer também aos meus amigos Lenin Abadie, Amanda Lima, Aldine Correia, Franklin Melo e Liza Minelli por todo apoio e compreensão.

Agradeço também ao meus pais Ailton e Lourdes, bem como minhas irmãs Daiane e Danielle por me entenderem e apoiarem.

Meu agradecimento mais que especial a Eduardo Vasconcelos, por todo amor, carinho, compreensão e paciência. ...

I open my eyes each morning I rise, to find a true thought, know that it's real, I'm lucky to breathe, I'm lucky to feel, I'm glad to wake up, I'm glad to be here, with all of this world, and all of it's pain, all of it's lies, and all of it's flipped down, I still feel a sense of freedom, so glad I'm around,

It's my freedom, can't take it from me, i know it, it won't change, but we need some understanding, I know we'll be all right.

—S.O.J.A. (Open My Eyes)

Resumo

Ao longo dos anos, sistemas computacionais em diversas áreas de conhecimento têm sido desenvolvidos e incorporados ao nosso cotidiano. Independente da etapa do ciclo de vida no qual o sistema se encontre é preciso preocupar-se com os requisitos de desempenho e dependabilidade dos serviços providos. Um dos métodos utilizados para estudos de desempenho é a medição de sistemas enquanto esses são submetidos aos diferentes níveis de carga possíveis. A aplicabilidade de tal técnica geralmente envolve o uso de programas de geração de carga sintética, que têm como intuito gerar eventos de carga de trabalho que simulem usuários requisitando serviços ao sistema. A carga de trabalho ocasionada por meio dessas ferramentas também pode contribuir para pesquisas de dependabilidade, visto que o uso de carga pode colaborar para revelar comportamentos não desejáveis e falhas não visualizadas anteriormente. O uso de carga sintética não é o único caminho a ser seguido para realizar estudos de dependabilidade. Ferramentas que geram situações de falha também podem ser empregadas para alcançar este objetivo.

A diversidade de funções para as quais sistemas são desenvolvidos requer que ferramentas específicas sejam construídas para a realização de experimentos de desempenho ou de dependabilidade, mesmo que haja um conjunto de conceitos comuns entre ferramentas deste tipo. Neste contexto, este trabalho propõe um *framework*, implementado em Java, que engloba vários métodos que podem auxiliar no desenvolvimento de ferramentas geradoras de eventos de forma mais ágil e confiável. No decorrer desta pesquisa geradores de carga, para experimentos de desempenho, e geradores de eventos de falha e reparo, para experimentos de dependabilidade, foram concebidos com o auxílio do *framework*, atestando, assim, sua eficiência. Por fim, estudos de caso desenvolvidos a partir dos *softwares* geradores de eventos serão apresentados.

Palavras-chave: avaliação de desempenho, dependabilidade, *framework*, carga de trabalho, injeção de falhas

Abstract

Over the years, computer systems in different areas of knowledge have been developed and incorporated in our daily activities. Regardless of the stage of the life cycle in which the system is in, there is the need of concerning about the requirements of performance and dependability of the provided services. One used method for measuring the performance is to subject the system to different load levels. The applicability of this technique usually involves the use of programs to generate synthetic load, which has the intention to generate workload events simulating users requesting services to the system. The workload applied by these tools can also contribute to dependability research, since the use of load can collaborate to reveal failures and undesirable behaviors not seen before. The use of synthetic load is not the only way to go for studies in dependability. Tools that generate fault situations can also be employed to achieve this goal.

The diversity of functionalities that systems are developed requires specific tools, which are built to perform experiments of performance or dependability, even if there is a set of common concepts between these kinds of tools. In this context, this paper proposes a framework, implemented in Java, which encompasses several methods that can assist in the development of tools that generate events in a more agile and more reliable way. During this research, load generators for performance experiments and fault and repair event generators for dependability experiments, were designed with the help of the framework, attesting thus its efficiency. Finally, case studies developed from software event generators will be presented.

Keywords: performance evaluation, dependability, framework, workload, fault injection

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

AoE	*ATA over Ethernet*
API	*Application Programming Interface*
AWS	*Amazon Web Services*
BIN	Número de Identificação Bancária
CGI	*Common Gateway Interface*
CSV	*Comma-Separated Values*
DDSFIS	*Debug-based Dynamic Software Fault Injection System*
DNS	*Domain Name System*
EBS	*Amazon Elastic Block Storage*
ECU	*EC2 Compute Units*
FITSEC	Fault Injection Tool based on Simulation and Emulation Cooperation
FPGA	*Field-Programmable Gate Array*
FTP	*File Transfer Protocol*
FuSe	*Fault injection using SEmulation*
GT-CWSL	*Georgia Tech Cloud Workload Specification Language*
GUI	*Graphical User Interface*
HDL	*Hardware Description Language*
HP	*Hewlett-Packard*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
IaaS	Infraestrutura como Serviço
IDE	*Integrated Development Environment*

IP	*Internet Protocol*
ISCSI	*Internet Small Computer System Interface*
JDBC	*Java Database Connectivity*
JVM	*Java Virtual Machine*
K-S	Kolmogorov-Smirnov
KVM	*Kernel-based Virtual Machine*
MTBF	*Mean Time Between Failure*
MTTF	*Mean Time to Failure*
MTTR	*Mean Time to Repair*
MVC	*Model-View-Controller*
NIST	*National Institute of Standards and Technology*
OSI	*Open Source Initiative*
PaaS	Plataforma como Serviço
PAC	*Presentation-Abstraction-Controller*
PARSIFAL	*Platform for Analysis and Reduction of Safety-critical Implementations*
PDV	Ponto de Venda
POS	*Point os Sale*
QDL	*Quasi Delay-Insensitive*
RAM	*Random Access Memory*
SaaS	*Software* como Serviço
SOAP	*Simple Object Access Protocol*
SSH2	*Secure Shell 2*
SWAT	*Session-based Web Aplication Tester*

SWORD *Scalable WORkloaD generator*

S3 *Simple Storage Service*

TCP *Transfer Control Protocol*

TEF Transferência Eletrônica de Fundos

UDP *User Datagram Protocol*

UML *Unified Modeling Language*

VHDL *VHSIC Hardware Description Language*

VSE *Virtual Server Environment*

WGCap *Workload Generator for Capacity Advisor*

WGSysEFT *Workload Generator for Electronic Funds Transfer System*

WS-FIT *Web Service - Fault Injection*

WOL *Wake on Lan*

1 Introdução

A mente que se abre a uma nova idéia jamais voltará ao seu tamanho original.

—ALBERT EINSTEIN

Este capítulo apresenta uma breve explanação sobre avaliação de desempenho e dependabilidade, destacando sua importância para desenvolver sistemas mais confiáveis e de bom desempenho. Em seguida são apresentados os objetivos deste trabalho, assim como sua motivação e as contribuições esperadas. Por fim, a estrutura do restante da dissertação é apresentada.

1.1 Contexto

A evolução dos sistemas computacionais é fruto do avanço em pesquisas que almejam oferecer aos usuários sistemas de fácil usabilidade, com funcionalidades diversificadas, confiáveis, de baixo custo e de alto desempenho. O cuidado empregado para que um sistema atenda estas características ocorre desde as fases iniciais do processo de desenvolvimento até sua implantação. Este mesmo empenho também é empregado no momento em que alguma atualização é necessária.

Dentre as características anteriormente mencionadas, necessárias a um sistema, duas tem atraído atenção não somente da comunidade acadêmica, mas também de administradores, projetistas e do usuário comum: desempenho e confiabilidade. Assegurar que os serviços providos sejam extremamente confiáveis e de alto desempenho tornou-se prioridade para sobrevivência dos sistema atuais, e um requisito fundamental para os novos sistemas. Portanto, estudos de avaliação de desempenho e dependabilidade podem e devem ser aplicados para obter estimativas do comportamento dos sistemas em execução.

Dessa forma, pode-se identificar, por exemplo, a melhor alternativa entre várias opções de projeto, e se melhorias no sistema são recomendáveis.

Técnicas de avaliação de desempenho podem ser empregadas em qualquer etapa do ciclo de vida do sistema (Jain, 1991). Entretanto, a fase do ciclo de vida na qual o sistema se encontra influencia na escolha da estratégia a ser aplicada para realização de estudos nesse âmbito. Notoriamente, a avaliação de desempenho possui três grandes conjuntos de técnicas (Obaidat and Boudriga, 2010): técnicas baseadas em modelagem analítica/numérica, técnicas baseadas em modelagem por simulação e técnicas baseadas em medição. Esta última é usada quando ao menos um protótipo do sistema está disponível, ou o próprio sistema já existe, ainda que nas primeiras versões (Jain, 1991; Fortier and Michel, 2002). Quando a medição não pode ser feita, então a modelagem analítica/-numérica, ou por simulação podem ser utilizadas. Entretanto, para alguns pesquisadores, os dados obtidos por meio dessas duas técnicas são mais convincentes quando baseados em medições anteriores.

As técnicas baseadas em medição envolvem a monitoração do sistema em teste quando este está sobre influência de uma carga de trabalho. Geralmente, os *softwares* empregados na geração de carga de trabalho são os chamados *Benchmarks*. *Benchmark* pode ser definido como o processo de executar um programa ou carga de trabalho em um sistema em específico e medir o desempenho resultante (Obaidat and Boudriga, 2010). Em geral, *benchmarks* estão dispostos em cinco categorias (Jain, 1991): adição de instrução, instrução *mix*, *kernel*, programas sintéticos e *benchmarks* de aplicação. Aplicações pertencentes a última categoria são usadas principalmente para avaliar o desempenho de sistemas quando destinados apenas a execução de um *software*, como reserva de passagens aéreas, por exemplo. Assim, fazem uso de quase todos os recursos computacionais, tais como: processador, rede, dispositivos de entrada e saída, entre outros.

O estudo de dependabilidade de sistemas também pode ser realizado por meio da ação de carga de trabalho. O uso de carga de trabalho pode contribuir para revelar falhas antes não perceptíveis, que possam existir em componentes do sistema. Uma outra forma de efetuar essas pesquisas é através da injeção de eventos de falhas, e observar o comportamento do sistema mediante seus efeitos. Dependabilidade é uma propriedade que integra atributos como disponibilidade, confiabilidade, segurança de funcionamento (*safety*), segurança (*security*), manutenabilidade, testabilidade e o comprometimento do desempenho (*performability*) (Pradhan, 1996). O emprego de alguns destes atributos em estudos contribui para que sistemas sejam justificadamente mais confiáveis.

Ferramentas geradoras de eventos, tanto de carga de trabalho como de falhas em

componentes de sistemas, podem ser aplicadas com a finalidade de conhecer os níveis de desempenho e dependabilidade dos serviços prestados sob diferentes cenários. Porém, um *software* concebido para atuar em teste em um determinado sistema, não necessariamente pode ser utilizado para agir em outro. Por exemplo, uma consulta a um sistema de banco de dados não acontece da mesma forma que uma transação bancária.

A diversidade de sistemas existentes, ou em desenvolvimento, impossibilita que uma ferramenta seja considerada padrão e adotada em todos os casos onde se precise avaliar questões de desempenho ou dependabilidade. Tal fato, faz com que equipes de desenvolvimento sejam incumbidas de criar ferramentas para medição de sistemas e testes de dependabilidade.

Desenvolver ferramentas geradoras de eventos não é uma tarefa trivial. Para concepção dessas ferramentas é preciso implementar um conjunto de métodos com a finalidade de estabelecer meios de comunicação com o sistema alvo, geração e controle de eventos, ou ainda, alguma maneira que permita que as solicitações ao sistema sejam de natureza aleatória, mas baseadas numa distribuição estatística conhecida. A realização dessa tarefa demanda tempo, visto que é necessário codificar todos os métodos necessários, testar, corrigir possíveis erros, e testar novamente, até que se obtenha o produto desejado.

1.2 Objetivos

O principal objetivo desse trabalho é propor um *framework* que englobe algumas das funcionalidades úteis ao desenvolvimento de ferramentas que visem oferecer suporte a estudos de desempenho e dependabilidade. Este trabalho almeja contribuir para que não haja retrabalho por parte do desenvolvedor ao codificar métodos que podem servir de alicerce para conceber ferramentas geradoras de eventos. De forma mais específica este trabalho se propõe a:

- Definir as funcionalidades geralmente empregadas quanto a protocolos de comunicação, criação e formas de gerenciamento de eventos para elaboração de ferramentas que visam apoiar estudos de desempenho e dependabilidade;
- Desenvolver um *framework* com base na definição de funcionalidades comuns às ferramentas de geração de carga de trabalho e de injeção de falhas;
- Construir uma ferramenta para teste de **desempenho** em um sistema específico, para validar e demonstrar a aplicabilidade do *framework* desenvolvido;

- Construir uma ferramenta para teste de **dependabilidade** em um sistema específico, para validar e demonstrar a aplicabilidade do *framework* desenvolvido; e
- Realizar experimentos de avaliação de desempenho e dependabilidade, utilizando as ferramentas concebidas através do *framework*.

1.3 Motivação

A ampla diversidade de sistemas computacionais, faz com que seja impraticável que um gerador de carga de trabalho possa ser aplicado em todos os casos de experimentos que envolvam estudos de desempenho ou dependabilidade. O desenvolvimento desse tipo de *software* mostra-se muitas vezes custoso devido à necessidade de se implementar todos os métodos que sejam necessários a seu funcionamento. Escrever os códigos para esta tarefa requer tempo, visto que este também segue o ciclo implementação-teste-ajuste-teste (Sommerville *et al.*, 2008), até que o objetivo esperado seja satisfeito. É importante observar que a construção de um gerador de evento é a atividade meio e não a atividade fim, visto que este esforço adicional significa o aumento do tempo de entrega do *software* alvo e consequentemente eleva o custo do desenvolvimento de *software*, entre outros problemas.

Baseado neste contexto, este trabalho foi motivado devido à necessidade de se desenvolver, de forma mais ágil e confiável, ferramentas de geração de eventos sintéticos que podem contribuir para a realização de experimentos nas áreas de avaliação de desempenho e dependabilidade. A junção de métodos referentes a protocolos de comunicação, criação e gerenciamento de eventos em um *framework* tenciona evitar que haja retrabalho por parte da equipe de desenvolvimento incumbida de criar geradores de eventos.

A adoção do *framework* proposto na construção de ferramentas geradoras de eventos pode ajudar na redução do número de iterações relacionadas ao ciclo de desenvolvimento destas ferramentas.

1.4 Contribuições Esperadas

O *framework* proposto neste trabalho, poderá beneficiar equipes de desenvolvimento que, necessariamente, precisem criar ferramentas de geração de eventos, para auxiliar em estudos de desempenho e dependabilidade. A adoção do *framework* em projetos de ferramentas geradoras de eventos visa contribuir para redução de tempo e esforço no processo de codificação. Isto ocorre devido ao *framework* fornecer um conjunto

de métodos já testados e validados por meio da construção de programas geradores de eventos sintéticos.

O *framework* proposto sugere ao desenvolvedor um conjunto de etapas a serem seguidas para criação de ferramentas geradoras de eventos. É importante mencionar que o *framework* não impõe ao desenvolvedor a adoção de todos os métodos sugeridos por este para criação de aplicações sintéticas. Assim, fica a cargo da equipe de desenvolvimento escolher quais métodos deseja utilizar.

1.5 Estrutura da Dissertação

O restante da dissertação está organizado como segue: o capítulo 2 apresenta a fundamentação teórica necessária ao desenvolvimento e compreensão deste trabalho. Nesse capítulo, é apresentado o conceito de *framework* e suas formas de desenvolvimento. Ainda neste capítulo, noções básicas sobre de avaliação de desempenho, dependabilidade, geração de carga de trabalho e técnicas de injeção de falhas são explanados. O capítulo 3 descreve brevemente alguns trabalhos relacionados, que são encontrados na literatura, sobre geração de carga e eventos de falhas com aplicabilidade no tema dessa dissertação. O capítulo 4 apresenta a principal contribuição deste trabalho: o *framework* denominado FlexLoadGenerator. Sua estrutura e modo de operação são descritos nesse capítulo. As ferramentas desenvolvidas tendo por base o *framework* serão apresentadas no capítulo 5, que também apresenta fundamentos sobre os sistemas para os quais tais ferramentas foram construídas. O capítulo 6 expõe a elaboração dos estudos de caso envolvendo as aplicações concebidas, assim como os resultados obtidos nos experimentos. Por fim, o capítulo 7 apresenta as considerações, limitações e dificuldades encontradas ao longo deste trabalho, sugerindo também alguns trabalhos futuros.

2

Fundamentos

Quem quer vencer um obstáculo deve armar-se da força do leão e da prudência da serpente.

—PINDARO

Este capítulo apresenta os principais conceitos desta dissertação. Inicialmente, são abordados os conceitos básicos sobre *frameworks* orientados a objetos, seguido dos princípios básicos de avaliação de desempenho e geração de carga de trabalho, destacando sua importância e aplicabilidade. Posteriormente, noções básicas sobre dependabilidade são apresentadas. Por fim, os principais fundamentos sobre as técnicas de injeção de falhas são discutidas, destacando-se as técnicas de injeção de falhas baseadas em *hardware* e *software*.

2.1 *Framework* Orientado a Objeto

Existem diversas definições de *frameworks*, no contexto de orientação a objetos. Segundo Zhang *et al.* (2008) *framework* pode ser definido como um conjunto de classes concretas e abstratas com seus relacionamentos e restrições, projetado para um domínio de aplicação específico. Enquanto para Mattsson and Bosch (1997) *framework* pode ser definido como um conjunto de classes que incorpora um projeto abstrato de soluções para uma família de problemas relacionados. Já Tomhave (2005) conceitua *framework* de uma forma bastante abrangente, como uma construção fundamental que define premissas, conceitos, valores e práticas, incluindo a orientação para implementação propriamente dita de um projeto de *software*.

Resumidamente, *framework* orientado a objeto pode ser entendido como um conjunto de classes concretas e abstratas que fornece uma implementação parcial de um

sistema ou parte deste para um dado domínio de problema. Diferentemente de uma biblioteca que contém um conjunto de classes que podem ser utilizadas separadamente um *framework* consiste de classes que possuem relacionamentos cujos os objetos interagem uns com os outros. Uma outra diferença entre bibliotecas e *framework* reside na forma de construção, enquanto bibliotecas são implementadas principalmente focando em reuso, *frameworks* não só reaproveitam códigos mais também partes significantes de um projeto.

Frameworks orientados a objetos, ou simplesmente *frameworks*, podem ser vistos como geradores, uma vez que se destinam a ser usados como alicerce para o desenvolvimento de uma série de aplicações pertencentes a um mesmo domínio de aplicação (Mattsson, 1996). Incorporados ao código, eles determinam a arquitetura da aplicação e predefinem os parâmetros de projeto, fazendo com que o desenvolvedor possa dedicar-se aos detalhes específicos do *software* em desenvolvimento (Carneiro, 2003). A Figura 2.1 ilustra a utilização de um *framework* na elaboração de sistemas ou sub-sistemas através de reuso.

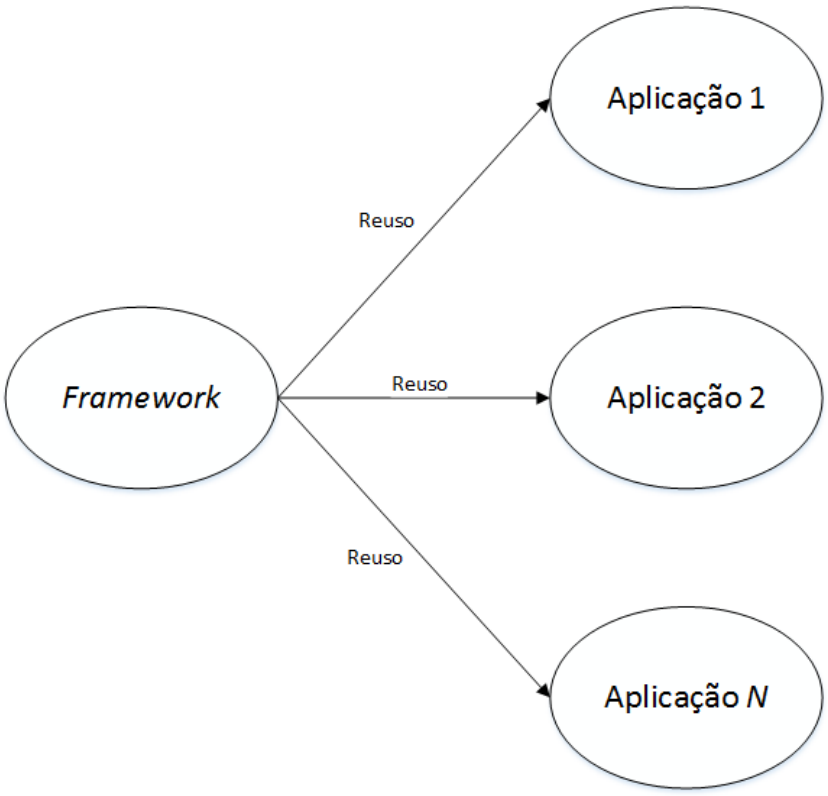


Figura 2.1 Representação de um *Framework* orientado à objeto. Fonte: Adaptado de (Mattsson, 1996)

O surgimento da programação orientada a objeto trouxe consigo alguns conceitos como herança, polimorfismo, *interface*, entre outros que tornaram possível a criação de *frameworks*. Estes elementos são básicos para construção dos chamados *hotspots* de um *framework*. Os *hotspots* são pontos de flexibilidade no *framework* que necessitam de complementação, onde os desenvolvedores adicionam o seu código especificando as funcionalidades do *software* em produção (Mattsson, 1996).

Um *framework* deve ser simples o bastante para que o desenvolvedor o compreenda e deve oferecer métodos flexíveis para que algumas de suas características possam ser alteradas através de redefinição de métodos, por exemplo. *Frameworks* são bastante atraentes por proporcionarem o desenvolvimento de sistemas de forma ágil, por meio do processo de reuso, frequentemente chamado de processo de instanciação (Penczek, 2008). A Figura 2.2 mostra a construção de uma aplicação a partir de um *framework*, onde no processo de instanciação, os *hotspots* são preenchidos de acordo com os requisitos específicos da aplicação.

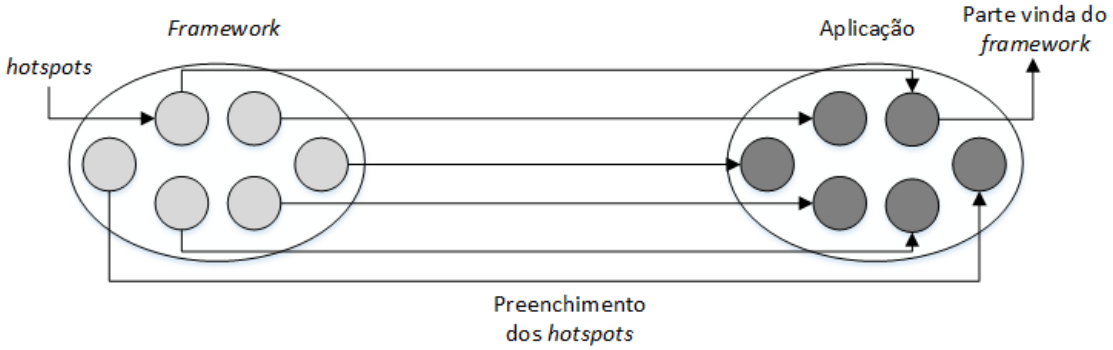


Figura 2.2 Instanciação de um *Framework* Orientado a Objeto. Fonte: Adaptado de (Penczek, 2008)

Segundo Mattsson (1996), os principais objetivos de um *framework* estão relacionados com manter o conhecimento da organização sobre o domínio da aplicação dentro da organização; minimização da quantidade de código necessário para implementar aplicações que pertençam ao mesmo domínio de problema; e possíveis adaptações às necessidades específicas das aplicações, como por exemplo, através de uma subclasse.

Frameworks orientados a objeto apresentam algumas vantagens e desvantagens. Entre as vantagens pode-se citar:

- Diminuição de linhas de código desenvolvidas se a funcionalidade requerida pela aplicação e a funcionalidade oferecida pelo *framework* pertencem ao mesmo domínio;

- O código do *framework* já esta escrito e depurado;
- *Frameworks* oferecem reuso de *design* e não somente código (Mattsson, 1996).

Entre as desvantagens de *frameworks* orientados a objetos pode-se citar:

- Dificuldades no desenvolvimento de *frameworks*, visto que experiência no domínio da aplicação é necessária;
- Difícil detecção de *hotspots*;
- Implementação e meios para validação de um *framework*.

Mesmo com algumas desvantagens a utilização de um *framework* no desenvolvimento de produtos de *software* pode ser importante, especialmente em grandes sistemas.

2.1.1 Caracterização de *Frameworks*

Um *framework* orientado a objeto pode ser caracterizado por diferentes dimensões. As mais importantes dimensões, segundo (Mattsson, 1996), são: o domínio do problema que o *framework* abrange, sua estrutura interna e como o *framework* pode ser utilizado.

Domínios de *framework* Três são os domínios de um *framework*: *framework* de aplicação, *framework* de domínio e *frameworks* de suporte. Os chamados *framework* de aplicação encapsulam funcionalidades que pode ser aplicados a diferentes domínios. Um exemplo deste tipo de *framework* são os *frameworks* para construção de *interfaces* gráficas. Os *framework* de domínio "capturam" o conhecimento e a experiência em um domínio de problema particular. Um exemplo deste tipo *framework* são *framework* para multimídia; e, por fim, os *frameworks* de suporte oferecem serviços para sistema de baixo nível como, por exemplo, *drivers* de dispositivo.

Estrutura de um *framework*

A estrutura interna de um *framework* está relacionado com os conceitos de arquiteturas de *software*. Mattsson (1996) descreve as arquiteturas de *frameworks* como:

- *Framework* arquitetural em camadas: ajuda a estruturar aplicações que podem ser decomposta em grupos de sub-tarefas com diferentes níveis de abstração;
- *Framework* arquitetural *Pipes* e *filters*: podem ser usados para estruturar aplicações que podem ser divididas em várias sub-tarefas totalmente independentes, que devem ser executadas em uma ordem seqüencial ou paralela;

- *Framework* arquitetural MVC (*Model-View-Controller*): define uma estrutura para aplicações interativas que separa a interface do usuário de seu núcleo funcional;
- *Framework* arquitetural PAC (*Presentation-Abstraction-Controller*): adequado para estruturar sistemas de *software* que são altamente interativos com usuários humanos;
- *Framework* arquitetural reflexivo: utilizado para aplicações que necessitam considerar uma futura adaptação às mudanças de ambientes, tecnologia e exigências, mas sem uma modificação explícita de sua estrutura e de implementação;
- *Framework* arquitetural *microkernel*: adequado para sistemas que oferecem diferentes pontos de vista sobre suas funcionalidades e, que muitas vezes, têm de ser adaptados às novas exigências de sistemas;
- *Framework* arquitetural *blackboard*: ajuda a estruturar aplicações complexas que envolvem vários subsistemas especializados para diferentes domínios; e
- *Framework* arquitetural *broker*: sistemas de *software* de estrutura distribuída em que os componentes interagem dissociados através de chamadas remotas de operação em um cliente/servidor.

Utilização de um *framework*

Um *framework* orientado a objeto pode ser utilizado de duas maneiras. Ou o usuário deriva novas classes a partir dele ou instancia e combina classes existentes.

A primeira abordagem é chamada de arquitetura dirigida ou herança focada. A abordagem principal é desenvolver aplicações baseando-se no mecanismo de herança. Os usuários do *framework* fazem adaptações neste através da derivação de classes e substituição de operações. A segunda abordagem (instanciação e combinação) é referida como *data-driven* ou composição focada em *frameworks*. A adaptação do *framework* as necessidades da aplicação implica no reaproveitamento da composição de um objeto. A forma como os objetos podem ser combinados fazem parte da descrição do *framework*, mas o que o *framework* faz depende de qual objeto o usuário passa para o *framework*.

Um *framework* dito *data-driven* é geralmente fácil de utilizar, porém é limitado.

2.1.2 Desenvolvimento de *Frameworks*

O desenvolvimento de um *framework* orientado a objeto requer um esforço considerável, mas os benefícios durante o processo de desenvolvimento de *software* geralmente

justificam o esforço inicial, visto que uma aplicação completa ou uma parte significante desta pode ser construída a partir de um *framework*. Construídos de forma a reutilizar tanto o código quanto as partes mais significantes de um projeto os *frameworks* podem ser implementados de duas formas. A primeira forma se baseia na profunda análise de domínio, enquanto a segunda forma é fruto do conhecimento adquirido durante o desenvolvimento de várias aplicações que pertençam a um mesmo domínio.

A primeira forma é bem semelhante ao desenvolvimento de *software*, iniciando com uma ampla pesquisa de domínio, entre outras coisas. Nesta forma de desenvolvimento exemplos de instâncias podem ser obtidos por meio de livros ou pessoas com amplo conhecimento do domínio estudado. Estas instâncias são coletadas e analisadas visando angariar partes comuns e partes que podem sofrer variações.

A segunda forma de desenvolvimento de um *framework* é relativa ao conhecimento adquirido pelos desenvolvedores durante o processo de implementação de vários *softwares* pertencentes a uma mesma família. A partir do conhecimento adquirido pode-se elaborar um *design* genérico separando as partes comuns e as específicas.

Ambas as formas de desenvolvimento citadas apresentam inconvenientes. Na primeira forma pode-se citar o custo de desenvolvimento pois, a primeira aplicação instanciada geralmente não é comercial e sim desenvolvida como uma forma de validação do *framework*. Enquanto na segunda forma, vícios de projeto de aplicações anteriormente desenvolvidas serão repassados as próximas aplicações instanciadas a partir do *framework*.

2.2 Avaliação de Desempenho

A Avaliação de desempenho é tida como uma arte (Jain, 1991). E como todo trabalho artístico, não pode ser produzido mecanicamente. Cada avaliação requer um grande conhecimento do sistema a ser modelado, além de uma cuidadosa seleção da metodologia, da carga de trabalho e das ferramentas empregadas. Avaliação de desempenho pode ser empregada, por exemplo, sempre que se deseja fazer comparações entre dois ou mais sistemas e encontrar o melhor para um dado conjunto de aplicações, decidir qual a melhor alternativa de *design* dentre várias opções, dentre outros (Jain, 1991; Lilja, 2000). Mesmo se não houver alternativas, avaliar o desempenho de sistemas pode contribuir para determinar a qualidade com que ele realiza suas atividades, e se melhorias são indicadas.

Experimentos envolvendo avaliação de desempenho de sistemas podem ser empregа-

dos em cada estágio do ciclo de vida de um sistema computacional (Jain, 1991; Obaidat and Boudriga, 2010). No entanto o ciclo no qual o sistema se encontra influencia na escolha da técnica a ser empregada para realização de estudos. Avaliação de desempenho possui três grandes conjuntos de técnicas (Obaidat and Boudriga, 2010): técnicas baseadas em modelagem analítica/numérica, técnicas baseadas em modelagem por simulação e técnicas baseadas em medição.

Técnicas baseadas em modelagem analítica/numérica descrevem o sistema de forma matemática (Lilja, 2000). Essas técnicas são geralmente utilizadas quando o sistema ainda não existe ou não está disponível. O tempo necessário para a obtenção de resultados faz desta técnica um atrativo devido a ocorrer de forma rápida. Um simples modelo analítico/numérico pode fornecer uma rápida visão do comportamento do sistema ou de um de seus componentes. Apesar de modelos analíticos serem aproximados, eles são aceitos, pois os próprios modelos podem ser usados para explorar alternativas de projeto. Isso é suficiente para estimar, aproximadamente, o comportamento e desempenho esperados. O tempo de construção de modelos é relativamente pequeno, assim como o custo empregado em seu desenvolvimento.

Técnicas baseadas em modelagem por simulação baseiam-se em modelos abstratos do sistema, logo podem ser utilizadas em qualquer etapa do ciclo de vida do sistema. Estas técnicas requerem um investimento considerável de tempo na derivação de modelos, concepção e codificação do simulador. O simulador pode ser facilmente modificado para estudar o impacto de mudanças feitas em qualquer componente do sistema simulado. A complexibilidade e o nível de abstração empregados em um simulador podem variar de acordo com o sistema. Com simulações, pode ser possível pesquisar o espaço dos valores dos parâmetros para combinação ideal. Modelagem por simulação é mais flexível, mais precisa, e possui maior credibilidade (Obaidat and Boudriga, 2010) quando comparado a modelagem analítica/numérica.

Ao contrário das técnicas baseadas em modelagem analítica/numérica e simulação, técnicas baseadas em medição só podem ser utilizadas quando algo semelhante ao sistema proposto já exista, mesmo que seja um protótipo (Jain, 1991). Dentre as técnicas apresentadas, esta é, sem dúvida, a que possui o maior custo, visto que requer equipamentos reais e tempo. Os resultados obtidos através desta técnica podem ser bastante variáveis pois os parâmetros definidos, assim como a carga de trabalho, podem influenciar no resultado dos experimentos. A credibilidade dos resultados obtidos através desta técnica são tidos como os mais confiáveis. Além disso, os resultados são, provavelmente, a maior justificativa quando consideradas as despesas envolvidas em sua utilização.

Às vezes é bastante útil utilizar duas ou mais técnicas ao mesmo tempo. Por exemplo, pode-se utilizar técnicas de modelagem analítica/numérica e simulação juntas na mesma avaliação para validar os resultados obtidos. Jain (1991) apresenta três regras de validação:

- Resultados obtidos por simulação não são confiáveis até que eles tenham sido validados por modelagem analítica/numérica ou por medição;
- Resultados obtidos através de modelagem analítica/numérica não são confiáveis até que eles tenham sido validados por simulação ou por medição; e
- Resultados obtidos por meio de medição não são confiáveis até que tenham sido validados por modelagem analítica/numérica ou por simulação.

A terceira regra mencionada é comumente ignorada. Medições são suscetíveis a erros, assim como as demais técnicas. O único requisito para validação é que os resultados não devem ser contra-intuitivos. Este método de validação é chamado de intuição de *expert*, e é comumente usada em modelos de simulação.

Duas ou mais técnicas podem ser utilizadas sequencialmente. Por exemplo, um modelo analítico simples pode ser usado para encontrar o intervalo adequado para os parâmetros do sistema e a simulação posteriormente utilizada para estudar o desempenho nesse intervalo. Isto reduz o número de execução de simulações consideravelmente e resulta numa utilização mais produtiva de recursos.

2.3 Medição de Desempenho

A medição de sistemas computacionais envolve essencialmente o monitoramento do sistema em estudo enquanto está sob influência de carga de trabalho. Para adquirir resultados expressivos, a carga de trabalho deve ser cuidadosamente selecionada. Esta carga pode ser real ou sintética (Fortier and Michel, 2002). A carga de trabalho real é observada em um sistema real em produção. Esse tipo de carga geralmente não é a mais adequada para realização de estudos de desempenho devido a não ser repetida (Jain, 1991), o tamanho da carga pode não ser considerável, os dados podem ter sofrido muitas perturbações, ou ainda, por questões de acessibilidade destes. Ao invés disso, uma carga sintética, cujas características são semelhantes a carga real, pode ser aplicada repetidamente e de forma controlada.

O principal motivo em se utilizar carga de trabalho sintética é o fato dela ser uma representação ou modelo da carga de trabalho real. Algumas outras razões para se utilizar a carga de trabalho sintética são a facilidade de modificação; a simplicidade de ser convertida para diferentes sistemas, devido ao seu pequeno tamanho; e a aplicação geradora de carga também poder ter embutida alguma capacidade de medição.

Na técnica de medição, diferentes tipos de métricas são geralmente necessárias, dependendo da natureza do sistema em teste (Obaidat and Boudriga, 2010). As diferentes estratégias de medição têm em sua base o conceito de evento, onde este é uma mudança no estado do sistema. A definição precisa de evento depende da métrica que esta sendo medida (Lilja, 2000). Por exemplo, um evento pode ser definido para ser uma referência à memória, acesso ao disco, comunicação de rede, uma mudança no estado interno do processador ou a combinação de outros sub eventos. Do ponto de vista do tipo de evento, estas métricas podem ser organizadas em uma das seguintes categorias (Lilja, 2000; Obaidat and Boudriga, 2010):

- **Métricas de contagem de evento:** esta categoria inclui as métricas que são simples contagem do número de vezes que um determinado evento ocorre. Por exemplo, quantidade de requisições de leitura/escrita em um disco;
- **Métricas de evento auxiliar:** registra valores de parâmetros secundários do sistema, sempre que um determinado evento ocorrer. Por exemplo, para determinar o número médio de mensagens na fila que são enviadas ao *buffer* de uma porta de comunicação, é necessário registrar o número de mensagens na fila cada vez que uma mensagem foi enviada ou removida da mesma. Assim, os eventos a serem monitorados serão o número de mensagens de entrada e saída da fila; e
- ***Profile:*** é uma métrica agregada utilizada para caracterização do comportamento global de um programa ou de um sistema inteiro. Normalmente, é usada para identificar onde o programa ou o sistema está gastando mais tempo de execução.

A estratégia utilizada para medir o desempenho da métrica de interesse pode ser decidida com base na classificação do tipo de evento discutido acima, onde as principais estratégias são (Obaidat and Boudriga, 2010):

- **Dirigida a evento:** faz o registro de informações necessárias para calcular a métrica de desempenho sempre que os eventos de interesse ocorrerem. Esta estratégia tem por vantagem que o *overhead* gerado ocorre apenas no registro de dados. Por

outro lado, se os eventos monitorados ocorrem com bastante frequência, então esta característica passa a ser uma desvantagem;

- ***Tracing:*** esta estratégia baseia-se na gravação de mais dados do que apenas um evento único. Isto resulta na necessidade de mais espaço de armazenamento, quando comparado a estratégia dirigida a evento;
- **Indireta:** esta estratégia é utilizada quando a métrica de interesse não pode ser medida diretamente. Neste caso, deve-se encontrar outra métrica que possa ser medida diretamente, da qual se pode deduzir a métrica desejada;
- **Amostra:** esta estratégia baseia-se na gravação do estado do sistema em intervalos de tempo a métrica de interesse. A frequência de amostragem determina o *overhead* de medição.

Algumas das formas de geração de carga de trabalho são discutidas na seção a seguir.

2.4 *Benchmark* e seus Tipos

Benchmark pode ser definido como o processo de executar um programa particular ou carga de trabalho em um sistema específico e medir o desempenho deste (Obaidat and Boudriga, 2010). *Benchmarks* são desenvolvidos visando um tipo particular de sistema ou um ambiente computacional. Este tipo de aplicação é utilizada para medir, prever o desempenho do sistema e revelar seu comportamento, bem como aspectos fortes e fracos. Além disso, estes *softwares* são criados obedecendo um conjunto de regras bem definidas quanto a gerenciamento de testes e procedimentos, incluindo entrada e saída de dados e medições de desempenho. Um *benchmark* pode ser um programa real que executa uma aplicação real ou uma aplicação sintética, que geralmente é projetada especificamente para exercitar certas unidades funcionais ou sub-unidades em várias condições de trabalho.

Programas *Benchmark* (*Benchmarks*) podem ser organizados em duas categorias: micro e macro *benchmarks* (Obaidat and Boudriga, 2010). Micro *benchmarks* medem um aspecto específico do sistema em teste, tal como memória, velocidade de dispositivos de entrada e saída, processador, aspectos de rede, entre outros. Contudo, macro *benchmarks* medem o desempenho do sistema como um todo. Sendo este último, importante para aplicações que requerem comparações entre sistemas ou alternativas de

design. Devido a isto, muitas vezes são usados para comparar diferentes sistemas com relação a uma determinada categoria de aplicação.

Uma outra classificação de *benchmarks* leva em consideração o tipo de aplicação como, por exemplo, computação científica, processamento de sinais, e assim por diante. Em geral, *benchmarks* usados para comparação de desempenho de sistemas podem ser genericamente classificados como (Jain, 1991): adição de instrução, *mix* de instruções, *kernel*, programas sintéticos e *benchmarks* de aplicação. Neste trabalho, serão discutidos apenas os programas sintéticos e os *benchmarks* de aplicação. Essas duas classificações serão discutidas nas subseções a seguir.

2.4.1 Aplicações Sintéticas

Com a proliferação de sistemas computacionais, testar dispositivos de entrada e saída também tornou-se uma parte importante para carga de trabalho. As primeiras tentativas de medir o desempenho de recursos de entrada e saída levaram os desenvolvedores a criar ferramentas que pudessem fazer um número específico de solicitações a serviços ou requisições de entrada e saída. A partir da aplicação dessas ferramentas puderam-se obter dados referentes ao tempo médio de utilização da CPU (*Central Processing Unit*), por exemplo. Por questões de portabilidade entre diferentes sistemas operacionais, geralmente estas ferramentas são escritas em linguagem de alto-nível.

Segundo Jain (1991) a primeira ferramenta deste tipo foi proposta por Buchholz (1969), que o chamou de programa sintético. Atualmente os programas sintéticos podem testar o desempenho de sistemas operacionais, aplicações, e também do *hardware*.

Algumas vantagens apresentadas por programas sintéticos são: (1) a facilidade de desenvolvimento; (2) não é necessário a utilização de arquivos contendo dados reais; (3) este tipo de aplicação pode ser facilmente modificada para atuar em um sistema diferente para o qual foi concebido. Além disso, algumas aplicações possuem a capacidade de medição. Assim, uma vez desenvolvido, o processo de medição pode ser automatizado e pode ser facilmente repetido em sucessivas versões de sistemas operacionais para obter os ganhos e perdas de desempenho.

2.4.2 *Benchmarks* de Aplicação

Benchmarks de aplicação são principalmente empregados para avaliar o desempenho de sistemas usados para hospedar apenas um produto de *software*, tais como: reserva de passagens aéreas, aplicações bancárias, e assim por diante. Para essas aplicações,

benchmarks são definidos como um grupo coletivo de funções, que fazem uso de quase todos os recursos do sistema, incluindo processadores, dispositivos de entrada e saída, redes e bancos de dados.

2.5 Dependabilidade

Dependabilidade é a capacidade de sistemas computacionais fornecer serviços que podem ser justificadamente confiáveis (Avizienis *et al.*, 2001). O serviço prestado por um sistema é o seu comportamento, tal como ele é percebido pelo usuário; o usuário em questão é outro sistema (físico ou humano) que interage com o primeiro na *interface* de serviço. Quando o sistema atende a especificação para o qual foi projetado, então o serviço é fornecido de forma correta. Uma falha de sistema é um evento que ocorre quando o serviço prestado se desvia do esperado, ou seja, é a transição do serviço correto para o incorreto. A transição do serviço incorreto para o correto é tido como restauração do serviço.

Uma definição alternativa de dependabilidade também apresentada por Avizienis *et al.* (2001) fornece o critério para decidir se um serviço é confiável ou não: a capacidade de um sistema em evitar falhas de serviço que são mais frequentes ou mais graves do que é aceitável, caso contrário, o sistema não é mais confiável.

Dependendo da aplicação destinada ao sistema, diferente ênfase pode ser colocada em diferentes facetas da dependabilidade, isto é, dependabilidade pode ser visto de acordo com diferentes, mas complementares, propriedades. Segundo Pradhan (1996) as propriedades de dependabilidade compreendem: disponibilidade, confiabilidade, segurança de funcionamento (*safety*), segurança (*security*), manutenabilidade, testabilidade e o comprometimento do desempenho (*performability*). Um resumo destas propriedades é mostrado a seguir.

A disponibilidade de um sistema computacional é a probabilidade de que ele esteja em atividade durante um determinado período de tempo, ou que tenha sido restaurado após a ocorrência de uma falha. Este atributo leva em consideração a alternância de períodos de funcionamento e reparo e geralmente é expresso sob forma de proporção de unidades de tempo, quando o serviço estava disponível e o período de serviço prestado (Trivedi *et al.*, 2009). A disponibilidade pode ser obtida através do cálculo:

$$A = \frac{uptime}{total_{time}} \tag{2.1}$$

Onde, *uptime* corresponde a soma de todos os intervalos de tempo onde o sistema

esta disponível, e $total_{time}$ é o tempo total de observação do sistema (*uptime* + *downtime*). Similarmente, a indisponibilidade é calculada como:

$$UA = \frac{downtime}{total_{time}} \tag{2.2}$$

Onde, *downtime* é a soma de todos os intervalos de tempo em que o sistema está indisponível. A disponibilidade também pode ser expressa em termos de número de noves conforme é apresentado na Equação 2.3, onde *100* representa o nível de disponibilidade máxima que o sistema pode atingir e A representa a disponibilidade do sistema.

$$N = 2 - log(100 - A) \tag{2.3}$$

Confiabilidade é a probabilidade de um sistema realizar as funções para o qual foi destinado sem a presença de falhas, durante um intervalo de tempo e de acordo com as condições concebidas. Estas condições implicam em (Weber, 2002):

- **Especificação:** sem uma especificação do sistema não é possível afirmar se o sistema está operando conforme previsto. Portanto, sem uma especificação formal e completa fica impraticável estabelecer se um sistema é ou não confiável;
- **Condições definidas:** as condições de funcionamento do sistema devem ser bem definidas;
- **Período de funcionamento:** o tempo de operação deve ser conhecido. Um sistema pode ser altamente confiável para uma determinada quantidade de horas de operação e depois necessitar de um período de reparo; e
- **Estado operacional no início do período:** não é possível falar em confiabilidade quando sistemas já iniciam seu funcionamento com defeitos.

A função de confiabilidade *R(t)* é a probabilidade de que o sistema irá funcionar corretamente, sem falhas, no intervalo de tempo de 0 a t e pode ser vista na Equação 2.4.

$$R(t) = P(T > t); t \geq 0 \tag{2.4}$$

Onde T é uma variável aleatória que representa o tempo de falha ou tempo para falha.

Segurança de funcionamento (*Safety*) é a probabilidade do sistema estar operacional e executar suas funções de forma correta ou descontinuar suas funções de maneira a não provocar danos a outros sistemas que dele dependam. *Safety* é a medida da capacidade

do sistema de se comportar de forma livre de falhas (*fail-safe*). Em um sistema *fail-safe*, ou a saída é correta ou o sistema é levado a um estado seguro (Weber, 2002).

Segurança (*Security*) implica na proteção contra falhas maliciosas, visando privacidade, autenticidade e integridade dos dados. Pode-se dizer que a segurança em sistemas está diretamente ligada à ausência de consequências catastróficas.

A manutenabilidade pode ser descrita como a probabilidade de que um sistema seja reparado após a ocorrência de um evento de falha em um determinado período de tempo, ou ainda sofrer eventuais modificações. Testabilidade é a capacidade de testar certos atributos internos ao sistema ou a facilidade em realizar testes. O comprometimento do desempenho (*performability*) descreve a degradação do desempenho de sistemas provocada pela ocorrência de falhas. Mesmo em decorrência de falhas o sistema continuará em operação, mas com degradações no nível de desempenho.

2.5.1 Medidas de Funcionamento

As medidas para avaliação de dependabilidade mais utilizadas na prática são: taxa de defeitos, MTTF (*Mean Time to Failure*), MTTR (*Mean Time to Repair*), e MTBF (*Mean Time Between Failure*). As medidas de MTTF e MTTR possuem relação com disponibilidade e confiabilidade (Fortier and Michel, 2002). Uma definição acerca destas medidas é apresentada a seguir:

- **Taxa de defeitos:** número esperado de defeitos de um componente em um dado período de tempo. Este período de tempo varia de acordo com o tempo de vida do componente (Weber, 2002).
- **MTTF:** tempo médio esperado até a primeira ocorrência de defeito;
- **MTTR:** tempo médio para reparação do sistema;
- **MTBF:** tempo médio entre defeitos do sistema.

Uma representação usual para a taxa de defeitos de componentes de *hardware* é dada pela curva da banheira apresentada na Figura 2.3.

Como pode ser visto na Figura 2.3, a curva da banheira apresenta três fases: taxa de falha decrescente, taxa de falha constante e taxa de falha crescente. Na primeira fase (taxa de falha decrescente), ocorre em um curto período de tempo em que geralmente a taxa de falha é bastante alta. Falhas ocorridas nesse período são ocasionadas por defeitos do processo de fabricação de equipamentos. Antes de disponibilizar o equipamento, os

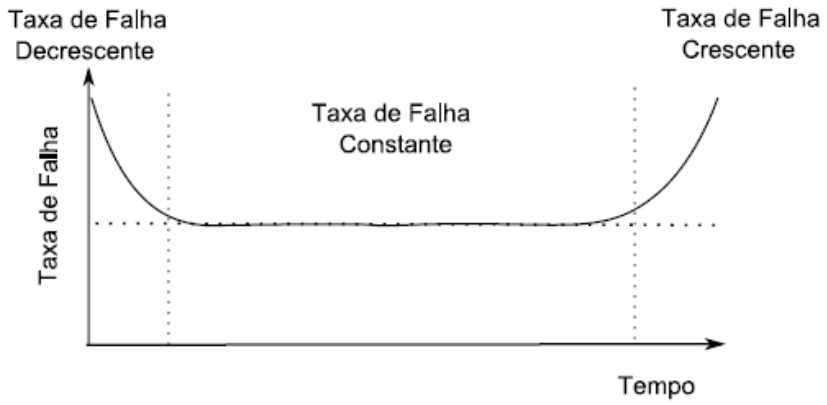


Figura 2.3 Curva da banheira. Fonte: Adaptado de (Ebeling, 2004)

fabricantes submetem estes a um processo chamado *burn-in*, onde eles são expostos a elevadas temperaturas de funcionamento. Este processo encurta o período de tempo de testes. Na segunda fase, conhecido por vida útil, as falhas ocorrem de maneira aleatória. Enquanto, durante a fase final, a taxa de falhas cresce exponencialmente. Isto ocorre porque o período de vida útil do equipamento normalmente não é constante; dependendo do nível de estresse em que o equipamento é submetido durante essa fase.

2.6 Técnicas de Injeção de Falhas

Sistemas computacionais necessitam manter o correto funcionamento mesmo na presença de falhas. No entanto, um sistema nem sempre realiza a função para o qual foi destinado de forma correta. As causas e consequências de desvios da operação esperada pelo sistema são chamados de fatores de dependabilidade, que compreendem a (Ziade *et al.*, 2004):

- **Falha:** é um defeito físico, imperfeição ou uma falha que ocorre dentro de algum componente de *hardware* ou *software*;
- **Erro:** é um desvio da precisão ou exatidão do sistema, sendo uma manifestação da falha. Uma falha durante o processamento pode levar a uma modificação do estado do sistema, o que é considerado um erro; e
- **Defeito:** é o não desempenho de alguma ação que era esperada pelo sistema.

Quando uma falha provoca uma mudança incorreta no estado do sistema, ocorre um erro. Embora a falha permaneça localizada no código ou no circuito, múltiplos erros

podem ser originados e se propagar pelo sistema. Quando mecanismos de tolerância a falhas detectam um erro, podem dar início a várias ações para lidar com as falhas e contê-las. Caso contrário, o sistema eventualmente apresenta mau funcionamento.

Há dois tipos de categorias de falhas: *hardware* e *software*.

- **Falhas de *hardware*:** podem surgir durante a operação do sistema e podem ser classificadas de acordo com sua duração, podendo estas ser em caráter permanente, transiente ou intermitente. As falhas permanentes são irreversíveis e geralmente causadas por danos a componentes; as falhas transientes são ocasionadas por condições específicas no ambiente, tais como interferência eletromagnética ou radiação, por exemplo; já as falhas intermitentes são causadas por instabilidade do *hardware*. Este último tipo de falha pode ser reparado por substituição de componentes, por exemplo.

- **Falhas de *software*:** são consequências de *design* incorreto, especificação ou tempo de codificação. No entanto, muitas dessas falhas estão latentes no código e ocorrem apenas durante a operação do sistema, especialmente sob alta ou incomum carga de trabalho. Uma vez que sendo resultado de má concepção, pode-se supor que todas as falhas de *software* sejam permanentes. Curiosamente, a prática mostra que, apesar de sua natureza permanente, o seu comportamento é transitório, isto é, quando um mau comportamento do sistema ocorre, ele não pode ser observado novamente, mesmo quando grande cuidado é tomado para repetir a situação em que ele ocorreu.

Algumas das falhas de *hardware* e *software* podem ser superadas quando descobertas precocemente, no processo de desenvolvimento ou em testes nos sistemas. As falhas não removidas podem afetar a dependabilidade do sistema.

A avaliação de dependabilidade é geralmente feita usando injeção de falhas. Injeção de falhas corresponde a inserir falhas de modo artificial em um sistema e avaliar seu comportamento diante das falhas inseridas (Vacaro and Weber, 2006). Falhas de *hardware* são facilmente injetadas por dispositivos de *hardware* destinados a esta tarefa ou por aplicações construídas com este propósito Ziade *et al.* (2004). Em ambos os casos perturbações podem ser causadas tais como, mudança de *bits* ou erros de *cache*, por exemplo. Neste caso, a escolha do método para injeção de falhas pode determinar o tipo de falha de *hardware*, seguindo a classificação descrita anteriormente. Falhas de *software* são geralmente a principal causa para interrupção do funcionamento de sistemas. Este tipo de falha pode se inserida no sistema através de programas injetores de falhas.

Estes programas podem ser construídos tanto para atingir sistemas operacionais como para afetar serviços prestados por aplicações específicas.

Para inserção de falhas em sistemas computacionais, tanto de *hardware* como de *software*, algumas técnicas podem ser empregadas. As técnicas de injeção de falhas, segundo Ziade *et al.* (2004), são classificadas em cinco categorias: injeção de falha baseada em *hardware*, injeção de falha baseada em *software*, injeção de falha baseada em simulação, injeção de falha baseada em emulação e injeção de falha híbrida. No entanto, neste trabalho serão discutidas apenas a injeção de falha baseada em *hardware* e a injeção de falha baseada em *software*. Estas duas técnicas são temas das subseções a seguir.

2.6.1 Injeção de Falhas Baseadas em *Hardware*

Injeção de falhas baseadas em *hardware* requer um *hardware* adicional para inserir falhas no sistema alvo. Este tipo de injeção de falhas permite que perturbações sejam inseridas no sistema em nível físico da máquina. Segundo Ziade *et al.* (2004), dependendo da falha e da localidade onde deve ser inserida, o método de injeção de falhas por *hardware* pode ser dividido em duas categorias:

- **Com contato:** o injetor tem contato físico com o sistema alvo introduzindo variação de corrente externamente ao *chip* do sistema em teste; e
- **Sem contato:** o injetor não possui contato físico direto com o sistema em teste. Nesta categoria, falhas são introduzidas por meio de fontes externas que produzem alguns fenômenos físicos como, por exemplo, interferência eletromagnética e radiação de íons.

Independente da categoria utilizada, alguns benefícios em utilizar a técnica de injeção por falhas baseadas em *hardware* são: o acesso a locais que não podem ser acessados por outras técnicas; experimentos são considerados rápidos, e; testes podem ser executados próximo ao tempo real. O uso desta técnica também acarreta alguns inconvenientes tais como: alto risco de dano no sistema em teste; baixa portabilidade e controle sobre falhas; além de reduzida observação do comportamento do sistema durante testes.

2.6.2 Injeção de Falhas Baseadas em *Software*

Injeção de falhas baseadas em *software* é uma abordagem mais flexível que a injeção de falhas baseadas em *hardware*, visto que permite simular falhas tanto em *hardware*,

como aplicativos e sistemas operacionais (Wanner, 2003). Esta técnica envolve a modificação do estado de funcionamento do *software* em execução no sistema em teste.

Ziade *et al.* (2004) divide esta técnica em duas categorias quanto as falhas inseridas: durante a compilação ou durante a execução do sistema. Na injeção de falhas durante a compilação as instruções do *software* devem ser modificadas antes da imagem do programa ser carregado e executado. Nesta categoria, falhas são injetadas através da alteração de código fonte ou no código de montagem da aplicação alvo. Assim, quando o programa modificado é executado as falhas são também acionadas. É importante observar que a alteração no código da aplicação pode inviabilizar o *software* de forma permanente.

Na inserção de falhas durante a execução é requerido um mecanismo adicional para introduzir falhas no sistema alvo. Esta categoria pode ser dividida de acordo com a forma em que as falhas são inseridas, são elas:

- ***Time-out:*** o mais simples dos métodos. Este método utiliza um temporizador (*hardware* ou *software*) para controlar a injeção de falhas. Quando termina o período de tempo predeterminado a falha é injetada;
- ***Exception/trap:*** neste caso, exceções de *hardware* ou *software* transferem o controle para o injetor de falhas, onde as falhas são inseridas sempre que um determinado evento ou condição ocorrer. Segundo Wanner (2003), a chamada ao injetor de falhas deve ser inserida no código fonte da aplicação alvo; e
- ***Code insertion:*** instruções são adicionadas ao *software* alvo para permitir que a inserção de falhas ocorra antes de determinados comandos. Ao contrário da categoria de inserção de falhas durante a compilação, este método apenas adiciona instruções ao sistema sem modificar o código já existente.

Alguns benefícios na escolha da técnica de inserção de falhas baseadas em *software* são: experimentos podem ser executados próximo ao tempo real; não requer *hardware* especial; e é de baixa complexidade e baixo custo de implementação. Em contrapartida, algumas dificuldades podem ser encontradas no uso desta técnica como: requerer modificações no código fonte; limitada observação e controlabilidade; e a não injeção de falhas em locais inacessíveis via *software*.

2.7 Considerações Finais

Este capítulo apresentou os principais conceitos desta dissertação. Primeiramente, foram apresentados conceitos básicos sobre *frameworks* orientados a objeto, avaliação de desempenho, medição de desempenho e *benchmarks* e seus tipos. Em seguida, os conceitos básicos sobre dependabilidade foram introduzidos. Por fim, os conceitos sobre técnicas de injeção de falhas foram apresentados.

3

Trabalhos Relacionados

Que os vossos esforços desafiem as impossibilidades, lembrai-vos de que as grandes coisas do homem foram conquistadas do que parecia impossível.

—CHARLES CHAPLIN

Este capítulo apresenta alguns trabalhos presentes na literatura que possuem relação com o tema de pesquisa abordado nesta dissertação. O presente capítulo foi dividido em seções para facilitar a leitura. Cada seção agrupa alguns dos trabalhos presente na literatura de acordo com o tema de pesquisa. Ao final de cada seção um quadro comparativo é apresentado para melhor observação do avanço da proposta desta dissertação em relação aos trabalhos correlatos.

Este trabalho utiliza a primeira forma de desenvolvimento de *frameworks* citada no capítulo 2 para implementar um *framework* que englobe algumas das funcionalidades úteis a criação de ferramentas geradoras de eventos de carga de trabalho ou de falha visando avaliação de desempenho ou dependabilidade de sistemas. Como será possível observar no decorrer deste capítulo os trabalhos existentes focam apenas em construir ferramentas de geração de eventos sintéticos, seja ele de carga de trabalho ou de falha, ou classificar algumas técnicas de injeção de falhas existentes, sem qualquer aproveitamento do conhecimento adquirido para construção de outras ferramentas de mesmo domínio.

3.1 Avaliação de Desempenho de Sistemas

Muitos trabalhos envolvendo a criação e utilização de carga de trabalho sintética para avaliação de desempenho dos mais variados tipos de sistemas, tem sido desenvolvidos

ao longo dos anos. Esta seção relata alguns dos trabalhos relacionados a avaliação de desempenho através do estresse do sistema em teste mediante a utilização de carga de trabalho. Como poderá ser observado ao longo desta seção, para cada sistema estudado equipes de desenvolvimento optaram por construir suas próprias ferramentas para auxiliar em testes de desempenho.

Galindo *et al.* (2009) apresenta a ferramenta WGCap (*Workload Generator for Capacity Advisor*), um gerador de carga de trabalho sintética voltado para o planejamento de capacidade em sistemas de servidores virtuais VSE (*Virtual Server Environment*), solução pertencente a HP (*Hewlett-Packard*). A ferramenta gera um *trace* (arquivo de registro) contendo dados de recursos computacionais como: demanda de CPU, memória, disco e rede. Diferente de algumas outras ferramentas de geração de carga de trabalho sintética, esta ferramenta não atua diretamente no sistema alvo. Seu objetivo é apenas gerar *traces* que possam ser importados por um dos componentes pertencentes ao VSE chamado de *Capacity Advisor*, que transforma os dados contidos no *trace*, efetivamente, em carga de trabalho.

Uma outra ferramenta para planejamento de capacidade de sistemas é apresentado por Patil *et al.* (2011). Chamada de VirtPerf, esta ferramenta é composta tanto por um gerador de carga de trabalho como por uma ferramenta que realiza as medições do sistema, capturando os níveis de utilização de recursos e o desempenho das métricas de aplicações que estão executando em circunstâncias controladas em ambientes virtualizados Xen ou KVM (*Kernel-based Virtual Machine*).

Anderson *et al.* (2006) apresenta a ferramenta SWORD (*Scalable WORkloaD generator*), um gerador de cargas para sistemas de processamento de dados distribuídos. A ferramenta em questão foi desenvolvida com o objetivo de realizar testes de desempenho em sistemas de processamento de dados, permitindo a geração de carga para uma variedade de aplicações e conteúdo.

Técnicas de geração de carga de trabalho para sistemas Web, como os abordados por Bahga and Madisetti (2011) e Krishnamurthy *et al.* (2006), sugerem formas de construir cargas para esse tipo de sistema, independente da aplicação que poderá estar contida nesses servidores. Bahga and Madisetti (2011) propõe uma técnica para gerar carga de trabalho de forma distribuída para aplicações que estiverem sendo executadas em ambientes de computação em nuvem, enquanto Krishnamurthy *et al.* (2006) aborda técnicas de teste de *stress* para geração de carga voltado a sistemas baseados em seção, na qual há dependências entre requisições. A abordagem proposta neste trabalho incluiu o desenvolvimento da ferramenta SWAT (*Session-based Web Aplication Tester*);

diferentemente de Bahga and Madisetti (2011) onde a técnica concebida tornou possível o desenvolvimento de uma linguagem chamada de GT-CWSL (*Georgia Tech Cloud Workload Specification Language*). A partir da aplicação de alguns outros parâmetros juntamente com essa linguagem é possível desenvolver geradores de carga de trabalho que atuem em ambiente de computação em nuvem. Embora os trabalhos apresentados por Bahga and Madisetti (2011) e Krishnamurthy *et al.* (2006) permitam elaborar ferramentas que injetam carga para avaliação de desempenho o domínio de concepção destas aplicações se limitam apenas a aplicações Web.

Panda *et al.* (2011) propõe uma metodologia de geração de carga de trabalho que visa contribuir para que desenvolvedores criem *benchmarks* personalizados para geração de carga de trabalho para arquitetura *multicore*. Para demonstrar a eficiência da metodologia os geradores ConWork e CompWork foram criados.

No âmbito de ferramentas *open-source* o JMeter (2013), é uma aplicação *desktop* que foi projetada para a realização de testes de desempenho e *stress* em aplicações cliente/-servidor, tais como aplicações Web. Ele pode ser usado para testar o desempenho tanto de recursos estáticos como dinâmicos tais como *servlets* Java, *scripts* CGI (*Common Gateway Interface*), objetos Java, banco de dados, entre outros. Por ser uma ferramenta inteiramente escrita em Java, o JMeter é compatível com ambientes capazes de suportar a máquina virtual Java. Esta ferramenta também permite a criação de testes para diversos protocolos, como HTTP (*Hypertext Transfer Protocol*), JDBC (*Java Database Connectivity*), FTP (*File Transfer Protocol*), SOAP (*Simple Object Access Protocol*), entre outros.

Schneider *et al.* (2005) faz uma explanação a respeito da geração de carga de trabalho em função de sistemas interconectados, com ênfase em multimídia. O foco deste trabalho é analisar o desempenho desses sistemas, mediante o uso de carga de trabalho.

A geração de carga de trabalho voltado a discos rígidos é o tema abordado por Ganger (1995), onde o trabalho mostra o desenvolvimento de uma abordagem para a validação de geradores sintéticos de requisições de disco rígido. Nesta abordagem uma série de dados estatísticos são retirados de requisições de disco rígido em operação para que cargas sintéticas possam ser geradas.

A questão de geração de carga para computação em *cluster* é abordado por Denneulin *et al.* (2004), onde este trabalho trata da questão de geração de carga de trabalho para sistemas que atuem de forma paralela. Este trabalho baseia-se na análise de um arquivo de *log* referente à utilização do I-cluster (*cluster* com 225 processadores), durante o período de um ano. A partir deste arquivo de *log* foi extraído um padrão no qual a geração

de carga foi baseada. Já a ferramenta StreamGen, concebida por Mansour *et al.* (2004), foi desenvolvida visando aplicações que possuem fluxo de informações distribuídas. O desenvolvimento do StreamGen contou com modificação de partes de outro gerador de carga denominado httperf (2013).

Busari and Williamson (2002) abordam o desenvolvimento e a utilização de um gerador de carga sintética para servidores *Web Proxy* chamado ProWGen. Este trabalho tem por objetivo investigar a sensibilidade das políticas de substituição da *cache* de servidores *Web Proxy* para cinco características de cargas Web selecionadas. Três políticas de substituição de *cache* são consideradas neste estudo, são elas: *least-recently-used*, *leastfrequently-used-with-aging* e *greedydual-size*.

Kant *et al.* (2001) apresenta a ferramenta Geist que tem por objetivo gerar carga para teste de *stress* voltado a servidores Web na forma de *e-commerce* e tráfego de internet. Este gerador é peculiar em relação a alguns outros de geradores de carga trabalho, pois, não se baseia na perspectiva do usuário para gerar carga e sim na forma como o servidor recebe a carga.

O trabalho apresentado por Van Ertvelde and Eeckhout (2010) se destina a discutir e comparar a respeito de redução de carga de trabalho e técnicas de geração que incluem: a redução de entradas, amostragem, mutação de código e síntese de *benchmark*. Neste trabalho maior ênfase é dado a mutação de código e síntese de *benchmark* devido aos autores considerarem estas técnicas menos conhecidas na época da publicação deste trabalho.

Diferentemente dos trabalhos descritos nesta seção, esta dissertação pretende reunir em um *framework* os principais meios para elaboração de ferramentas de geração de eventos tanto para avaliação de desempenho como para dependabilidade. Porem, na presente seção foram debatidos apenas trabalhos relacionados a construção de ferramentas visando avaliar o desempenho de sistemas. Dentre os trabalhos apresentados alguns se destacam dos demais devido a sua relevância. Estes trabalhos estão expostos na Tabela 3.1 onde, destacou-se pontos como: qual o sistema alvo dessas ferramentas, pontos fortes e fracos, além de sua contribuição.

Vale salientar que nenhum dos trabalhos apresentados permite a construção de uma família de aplicações de mesmo domínio, com exceção dos trabalhos apresentados por Krishnamurthy *et al.* (2006) e Bahga and Madisetti (2011). Estes dois trabalhos proporcionam a criação de ferramentas com características específicas, mas, para isto, é necessário a construção de modelos e/ou a utilização de *logs*, onde a utilização de *logs* faz com que a ferramenta só possa ser construídas para sistemas já em produção.

Os trabalhos correlatos que discursam sobre técnicas e ferramentas de injeção de falhas, pertencentes a área de dependabilidade, é apresentada na seção a seguir.

Tabela 3.1 Quadro com alguns dos trabalhos pertencentes a área de avaliação de desempenho

Ferramenta	**Sistema alvo**	**Ponto(s) forte(s)**	**Ponto(s) fraco(s)**	**Contribuição**
WGCap	Servidores virtuais VSE	(1) Geração de traces a partir de diversas distribuições de probabilidade; e (2) Importação de *trace* existente para geração de outro similar	(1) Específica para servidores virtuais VSE; (2) Não faz comunicação com o sistema alvo	Planejamento de capacidade em servidores virtuais VSE
VirtPerf	Ambientes virtualizados	(1) Gera carga e realiza medições do sistema em estudo; e (2) Execução em ambiente controlado	Se limita aos ambientes Xen ou KVM	Avaliação de desempenho em ambientes virtualizados
SWAT	Sistemas baseados em seção	(1) Testa sistemas com dependência entre requisições; e (2) Cria ferramentas com as características específicas solicitadas	Utiliza de modelos e *logs* para construção de ferramentas	Criação de ferramentas para teste de desempenho
GT-CWSL	Ambientes em nuvem	Criação de ferramentas de geração de carga de trabalho	Utiliza de modelos para concepção de ferramentas	Criação de ferramentas para teste de desempenho
Jmeter	Diversos	(1) Geração de carga para diversos tipos de sistemas; (2) Pode ser utilizado tanto para testar recursos estáticos como dinâmicos; (3) Criação de testes para diversos protocolos; e (4) Novos protocolos podem ser adicionados	Adição de novas funcionalidades requer um amplo estudo para que esta possa ser criada e incorporada ao JMeter	Avaliação de desempenho de diversos tipos de sistemas

3.2 Avaliação de Dependabilidade de Sistemas

Esta seção apresenta alguns trabalhos envolvendo técnicas de criação e injeção de falhas objetivando avaliar a dependabilide de sistemas. A introdução de falhas em sistemas contribui para avaliar duas das propriedades de dependabilidade, disponibilidade e confiabilidade. No decorrer desta seção será possível observar o emprego de diferentes técnicas na construção de ferramentas ou ambientes que emulem injeção de falhas em

sistemas.

Ziade *et al.* (2004) descreve técnicas de injeção de falhas. Este trabalho apresenta cinco técnicas principais: (1) injeção de falhas baseadas em *hardware*; (2) injeção de falhas baseadas em *software*; (3) injeção de falhas baseadas em simulação; (4) injeção de falhas baseadas em emulação e, por fim, (5) injeção de falhas híbridas. Neste trabalho também são realizados algumas comparações sobre qual a melhor técnica a ser empregada dado um determinado problema, além de apresentar algumas ferramentas presentes na literatura que se enquadram de acordo com cada técnica apresentada, sua descrição e forma de atuação, além dos pontos fortes e fracos.

Zhang *et al.* (2011) apresenta um injetor de falhas para testes de dependabilidade com atuação em sistemas embarcados de tempo real denominado DDSFIS (*Debug-based Dynamic Software Fault Injection System*). A metodologia empregada para geração de eventos de falhas conta com: a modificação de valores armazenados em variáveis, utilização de meios como *jump* de instruções e o cancelamento de execução de funções do sistema prematuramente. Estes tipos de perturbações fazem com que as falhas geradas por essa ferramenta não sejam permanentes. Desenvolvido através da utilização de técnicas de injeção de falhas baseadas em *software*, esta ferramenta detecta de forma automática os locais onde falhas devem ser injetadas, sem a necessidade de recompilar o sistema alvo.

Uma outra ferramenta também baseada em *software* é apresentada por Looker *et al.* (2004). Denominada WS-FIT (*Web Service – Fault Injection*), esta ferramenta injeta falhas a nível de rede para testar sistemas baseados no protocolo SOAP através de modificações de mensagens SOAP quando assinatura e encriptação estão sendo usados.

Um conjunto de ferramentas para avaliação de dependabilidade, denominado DS-Bench Toolset, é apresentado por Fujita *et al.* (2012). O DS-Bench é composto por: D-Case Editor, DS-Bench e D-Cloud, onde, juntos, estes módulos provêem a obtenção de métricas de dependabilidade do sistema alvo através do uso de *benchmarks*, no qual estes *benchmarks* testam o sistema por completo (sistema operacional, rede, dentre outros) em um ambiente contendo tanto máquinas físicas como virtuais.

Cao *et al.* (2012) apresentam uma revisão crítica a respeito de dois diferentes níveis de análise do comportamento de falhas, classificados em: método de análise do comportamento de falha de elementos e método de análise do comportamento de falhas do sistema. Enquanto Zhang (2010) apresenta uma pesquisa sobre tecnologia de injeção de falhas de *software* baseada em sistemas distribuídos.

No campo de injeção de falhas baseadas em *hardware*, Friesenbichler *et al.* (2010)

apresenta uma abordagem que utiliza circuitos assíncronos QDL (*Quasi Delay-Insensitive*) para injeção de falhas de *hardware*, onde as falhas são causadas por meio de sabotadores configuráveis colocados em pontos específicos do circuito. O trabalho apresentado por Svenningsson *et al.* (2010) aborda como modelos de injeção de falhas podem ser utilizado para simular o efeito de falhas de *hardware* em sistemas embarcados. Embora utilizando-se da técnica de simulação para injeção de falhas de *hardware*, este trabalho também realiza comparações a nível do modelo e a nível de *hardware* usando Simulink e um microcontrolador Infineon, respectivamente.

Jeitler *et al.* (2009) apresenta o FuSe (*Fault injection using SEmulation*), uma ferramenta injetora de falhas que combina o desempenho de um protótipo implementado em *hardware* e a flexibilidade, bem como a visibilidade de uma simulação em HDL. Devido ao FuSE ser baseado no SEmulator FPDA falhas podem ser injetadas tanto em modelos HDL (*Hardware Description Language*) ou em um *netlist* baixado para um FPGA (*Field-Programmable Gate Array*), onde a comutação entre estes dois modos é realizada de forma transparente, permitindo a observação da propagação de falhas no circuito.

Weinkopf *et al.* (2006) propõe um modelo de falha não clássica para emulação de falhas e um ambiente genérico que suporta diferentes tipos de modelos. O ambiente de nome PARSIFAL (*Platform for Analysis and Reduction of Safety-critical Implementations*) apresentado neste trabalho e pode ser adaptado para executar diferentes emulações em sistemas. Já Baraza *et al.* (2000), apresenta o protótipo de uma ferramenta automática e independente de modelo de injeção de falhas baseado em VHDL (*VHSIC Hardware Description Language*).

Ejlali *et al.* (2003) demonstra o uso da técnica de injeção de falhas de modo híbrido através da cooperação entre as técnicas de simulação e emulação de falhas. Neste trabalho é apresentada a ferramenta intitulada FITSEC (*Fault Injection Tool based on Simulation and Emulation Cooperation*), baseada em Verilog e VHDL. Esta ferramenta foi desenvolvida com o objetivo de apoiar todo o processo de concepção de sistemas.

Diferentemente dos trabalhos anteriormente mencionados nesta seção o trabalho apresentado por Zhou *et al.* (2011) visa avaliar o impacto causado ao desempenho de sistemas de larga escala frente ao atraso na reparação de falhas. Os sistemas analisados neste estudo são o IBM Blue Gene e um *cluster*, onde a análise foi proporcionada através de uma carga real coletada ao longo do funcionamento do supercomputador.

Uma das utilidades do *framework* proposto nesta dissertação é prover meios para construir ferramentas de geração de eventos de falhas utilizando a técnica de injeção de falha baseada em *software* discutida no capítulo 2. Nenhum dos trabalhos apresentados

nesta seção fornece capacidade para criação de ferramentas, limitando-se apenas a gerar e injetar um conjunto de falhas no sistema em estudo ou simular o sistema alvo mediante a ocorrência de falhas e observar seus efeitos sobre a dependabilidade. Alguns dos trabalhos correlatos descritos anteriormente estão presentes na Tabela 3.2, juntamente com o sistema alvo, pontos fortes e fracos e a principal contribuição.

Tabela 3.2 Quadro com alguns dos trabalhos pertencentes a área de dependabilidade

Ferramenta	**Sistema alvo**	**Ponto(s) forte(s)**	**Ponto(s) fraco(s)**	**Contribuição**
DDSFIS	Sistemas embarcados de tempo real	(1) Detecção de locas para injeção de falhas de forma automática; e (2) Não recompilação do sistema alvo	Apresenta apenas três tipos de falhas: modificação de valor de variáveis, *jump* de instruções e cancelamento de execução de funções do sistema	Testes de dependabilidade através de injeção de falhas de *software*
WS-FIT	*Web service* que utilizam o protocolo SOAP	Modificações de mensagens baseadas em protocolo SOAP com encriptação e assinatura	Limitado a troca de mensagens via protocolo SOAP	Injeção de falhas em redes que utilizam SOAP
DS-Bench Toolset	Diversos	(1) Obtém as métricas que serão utilizadas em testes de dependabilidade; (2) Teste completo do sistema alvo; e (3) Ferramenta aplicada tanto em máquinas físicas como virtuais	Utiliza *benchmarks* para obtenção das métricas de dependabilidade	Avaliação de dependabilidade para máquinas físicas e virtuais
PARSIFAL	Diversos	Utilização de modelos de falhas não clássica	Realiza apenas emulações de falhas	Emulação de falhas utilizando modelos

3.3 Considerações Finais

Este capítulo apresentou alguns dos trabalhos relacionados presente na literatura nas áreas de avaliação desempenho e dependabilidade de sistemas. Inicialmente, na seção 3.1, foi apresentado alguns dos trabalhos envolvendo o desenvolvimento e aplicação de ferramentas de geração de carga de trabalho sintética para avaliar o desempenho de

sistemas. Alguns trabalhos explanam a respeito de ferramentas de geração de carga para sistemas específicos, outros abordam ferramentas que são capazes de conceber outras ferramentas pertencente a um domínio. No entanto, essas ferramentas requerem o uso de alguns artifícios como a utilização de *benchmarks*, modelos ou mesmo *logs* de sistemas em produção, o que nem sempre é desejável.

Em seguida, na seção 3.2, foram expostos alguns trabalhos relacionados a técnicas de injeção de eventos de falhas para avaliar a dependabilidade de sistemas, frente a adversidades causadas por falhas. Os trabalhos nesta área focam na utilização de uma ou a combinação de técnicas de inserção de falhas ou ainda discutir e classificar as técnicas existentes. Entretanto, nenhum dos trabalhos apresenta injeção de falhas em aplicações específicas ou a possibilidade de criar ferramentas de uma mesma família, ou ainda explorar com mais afinco as possibilidades que a inserção de falhas baseadas em *software* pode oferecer.

Como será apresentado, esta dissertação pretende utilizar dos conhecimentos obtidos neste capítulo para elaboração e implementação de um *framework*. Este *framework* poderá auxiliar na construção de ferramentas com intuito de avaliar o desempenho de sistemas por meio de uso de carga de trabalho ou verificar algumas das propriedades de dependabilidade através de inserção de falhas baseadas em *software* no sistema em teste.

4

Infraestrutura de Geração de Eventos - FlexLoadGenerator

Suba o primeiro degrau com fé. Não é necessário que você veja toda a escada. Apenas dê o primeiro passo.

—MARTIN LUTHER KING

Este capítulo é dedicado a apresentar o *framework*, denominado FlexLoadGenerator, foco deste trabalho. Inicialmente uma visão geral sobre a problemática abordada nesta pesquisa é apresentada. Logo após é descrita a metodologia empregada no processo de desenvolvimento do *framework*, seguido pelos diagramas UML (*Unified Modeling Language*) (Fowler, 2004) que representam graficamente o FlexLoadGenerator, auxiliando sua compreensão. Ainda neste capítulo, são abordados o núcleo de geração de números aleatórios e a descrição do *framework*. Em seguida, é apresentado um exemplo prático do emprego do *framework* na construção de ferramentas geradoras de eventos. Por fim, são descritos os testes dos métodos pertencentes ao *framework*.

4.1 Visão Geral

O FlexLoadGenerator é um *framework* para construção de ferramentas geradoras de eventos para estudos nas áreas de avaliação de desempenho e dependabilidade de sistemas. Desenvolvido em Java (Deitel and Deitel, 2010), o *framework* possui alguns métodos que podem vir a auxiliar no desenvolvimento de programas de geração de eventos sintéticos, tais como meios de comunicação, criação e controle de eventos. Assim, fica a cargo da equipe de desenvolvimento elaborar qual evento será produzido, e escolher como a aplicação sintética deve interagir com o sistema alvo. A flexibilidade é

favorecida por intermédio da escolha de uma dentre as opções que o FlexLoadGenerator disponibiliza para criação e controle de eventos.

Quando o desenvolvedor opta por utilizar a estrutura do FlexLoadGenerator para criação e gerenciamento de eventos é preciso seguir um ciclo para concepção de ferramentas. O fluxo de criação de aplicações de geração de eventos sintéticos será descrito mais adiante, neste capítulo.

Alguns termos próprios de programação de *software* orientado a objeto serão constantemente encontrados ao longo deste capítulo, como: classe, classe abstrata, construtor, herança, instância, laço de repetição, método, método abstrato, objeto, pacote, subclasse, superclasse, *thread*, dentre outros. As definições destes termos não será discutida nesta dissertação devido a essa nomenclatura ser considerada padrão em projetos que envolvam programação orientada a objetos. O significado de cada termo pode ser encontrado com riqueza de detalhes em (Deitel and Deitel, 2010; Sebesta, 2003).

4.2 Desenvolvimento do *framework*

Uma vez que a problemática foi definida e as funcionalidades que o *framework* deveria conter foram escolhidas, iniciou-se o processo de concepção do FlexLoadGenerator. A sequência de passos seguidos para atingir o objetivo deste trabalho é ilustrado pelo fluxograma presente na Figura 4.1.

A primeira atividade do fluxograma trata da escolha da linguagem de programação utilizada no desenvolvimento do *framework*. Tendo em vista a abrangência, a flexibilidade e a reusabilidade que o *framework* foi idealizado, optou-se por utilizar a linguagem Java (Java, 2013). Um outro aspecto que influenciou na escolha dessa linguagem foi o fato da mesma permitir a construção de vários tipos de aplicativos, sejam eles móveis, corporativos, dentre outros. Uma das características mais importantes do Java é a sua portabilidade: uma aplicação pode ser executada em plataformas e sistemas operacionais distintos, desde que suporte a máquina virtual Java (JVM).

A segunda atividade do fluxograma refere-se a escolha do ambiente de programação utilizado para codificar o *framework*. Durante esta etapa o IDE (*Integrated Development Environment*) Eclipse (Eclipse, 2013) foi selecionado.

A terceira atividade no fluxograma compreende a definição da arquitetura. Nesta etapa foi definida e documentada a arquitetura do FlexLoadGenerator com o auxílio de diagramas de classes e de sequência. Esta etapa é extremamente importante pois, a mesma, orientou o passo a passo a ser adotado durante o processo de implementação do

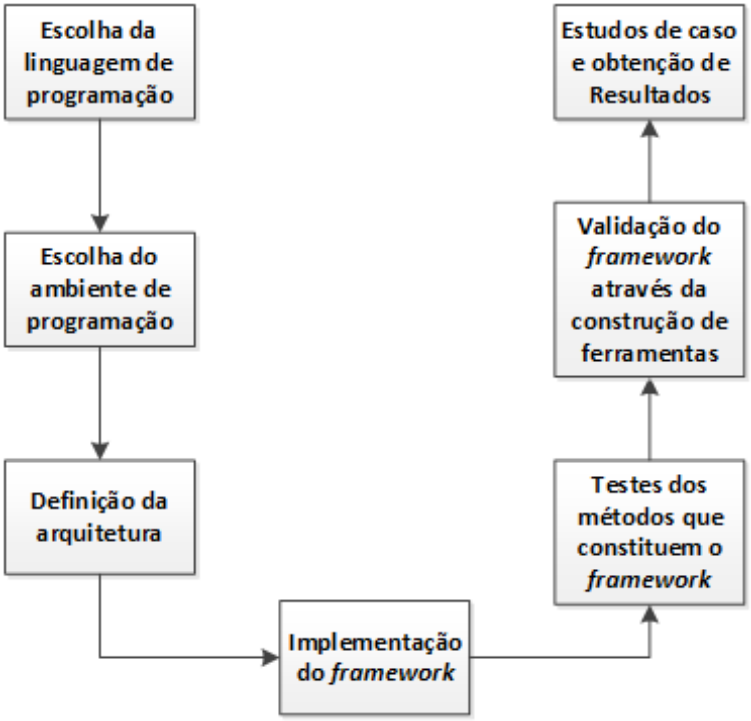


Figura 4.1 Fluxograma das atividades adotadas durante a implementação do FlexLoadGenerator

framework. Em uma atividade posterior a documentação foi ampliada com a adição do Javadoc.

A quarta atividade do fluxograma trata da implementação do *framework*. Nesta etapa ocorreu toda a parte de codificação referente ao FlexLoadGenerator onde, através do ambiente de programação Eclipse, foram criadas todas as classes e métodos selecionados para integrar o *framework*.

A quinta atividade corresponde ao teste dos métodos implementados. Nesta etapa todos os métodos que compõem o *framework* foram testados objetivando verificar se estes desempenhavam as atividades para o qual foram criados, conforme o esperado.

A sexta atividade é relacionada a validação do *framework*. Esta atividade é essencial pois visa analisar a efetividade do *framework* através da redução no esforço de codificação de ferramentas. Nesta atividade duas ferramentas com intuito de criar eventos de carga ou de causar falhas em sistemas foram criadas. O Capítulo 5 apresenta as ferramentas desenvolvidas em maiores detalhes, além dos esforços de codificação para concebê-las.

A sétima e última atividade presente no fluxograma, é voltada a analisar a eficiência das ferramentas construídas a partir do FlexLoadGerator por meio dos resultados obtidos

em estudos de caso, onde estes envolvem avaliar o desempenho de sistemas frente a uma grande quantidade de carga de trabalho ou a disponibilidade do sistema frente a ocorrência de falhas. O Capítulo 6 descreve em maiores detalhes como esta etapa foi realizada.

4.3 Diagramas UML e Documentação

Diagramas UML (*Unified Modeling Language*) são amplamente utilizados para modelagem de sistemas construídos com o paradigma de orientação a objetos (Sommerville *et al.*, 2008; Fowler, 2004). Dentre os diagramas UML, utilizou-se apenas dos diagramas de classes e sequência devido ao diagrama de classe ser uma representação gráfica estruturada que descreve as classes e seus relacionamentos, e o diagrama de sequência retratar as interações entre os objetos. Os demais diagramas UML, como de atividade, de casos de uso e de estado, por exemplo, não foram utilizados por considerarmos que os diagramas de classes e sequência cobriram todo o entendimento necessário para utilização do *framework*.

4.3.1 Diagrama de Classe

A Figura 4.2 apresenta o diagrama de classes do FlexLoadGenerator. Optou-se por não exibir no diagrama as classes *AgentClassNotProvidedException* e *WrongAmountOfParametersException*, usadas para tratamento de exceções por não possuírem qualquer relacionamento com as demais. Do mesmo modo, não estão presentes no diagrama os atributos e métodos pertencentes as classes, em razão do reduzido tamanho da fonte (letra) que, neste formato, poderia inviabilizar a leitura da dissertação quando impressa. O diagrama completo, incluindo as classes relacionadas a tratamento de exceções, está disponível no Apêndice A.

O diagrama de classes retratado na Figura 4.2, apresenta uma perspectiva na qual são abstraídos detalhes de desenvolvimento. Neste diagrama é possível observar cada classe, o relacionamento entre elas, se são classes concretas ou abstratas, além de questões como herança, entidade forte e entidade fraca.

Como pode-se observar no diagrama, a classe *ClientAgent* é uma classe abstrata, assim como as classes *ClientAgentUDP*, *ClientAgentSSH* e *ClientAgentTCP*. As demais classes do diagrama são concretas. Todas elas pertencem ao mesmo pacote (*generator*), conforme o diagrama posiciona essa informação entre "{ }" logo abaixo do nome de cada classe. A seta aberta, sem qualquer preenchimento, indica que as classes *ClientA-*

gentUDP, *ClientAgentSSH* e *ClientAgentTCP* possuem um relacionamento com *ClientAgent*, no qual herdam seus atributos e métodos.

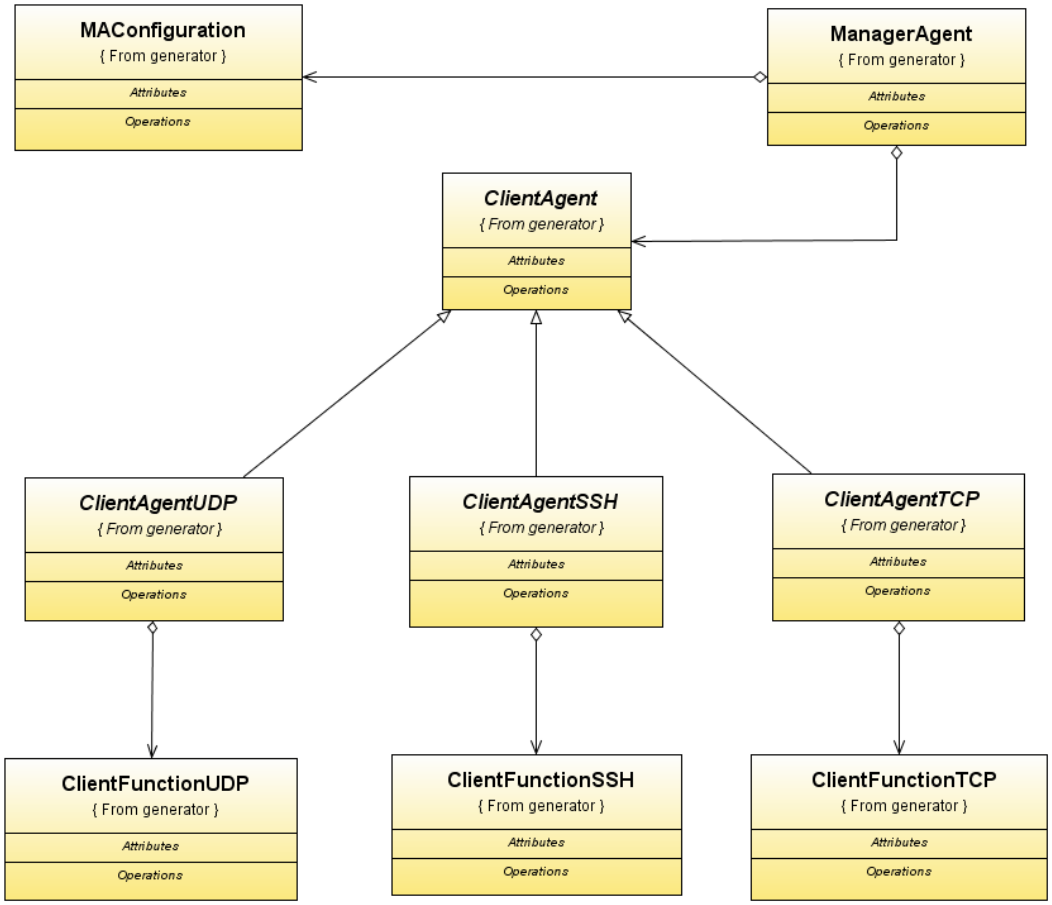


Figura 4.2 Diagrama de classes FlexLoadGenerator

A representação das setas, apenas com traços, sem preenchimento, mostra a navegabilidade (forma de leitura) entre essas classes. Os losangos que tocam algumas das classes indicam o relacionamento de agregação "todo-parte", no qual a parte não pode existir sem o todo, além de demostrar quem é a entidade forte presente no relacionamento.

4.3.2 Diagrama de Sequência

Além do diagrama de classes, foram elaborados diagramas de sequência a fim de permitir melhor compreensão do funcionamento do FlexLoadGenerator. Estes diagramas apresentam as mensagens trocadas entre objetos quando ferramentas de geração de eventos sintéticos são construídas utilizado o *framework*. Os diagramas de sequência

apresentados nas Figura 4.4, Figura 4.5, Figura 4.6 e Figura 4.7 completam o diagrama de sequência apresentado na Figura 4.3, pois estes diagramas são as possíveis formas de execução para o método *generateLoad()* presente na Figura 4.3, retratando assim as possíveis formas de como uma ferramenta de geração de eventos pode ser projetada para funcionar.

A descrição e utilidade das classes e métodos expostos nos diagramas de sequência desta seção pode ser visto no Apêndice B. A forma de leitura de diagramas de sequência pode ser encontrada em (Sommerville *et al.*, 2008).

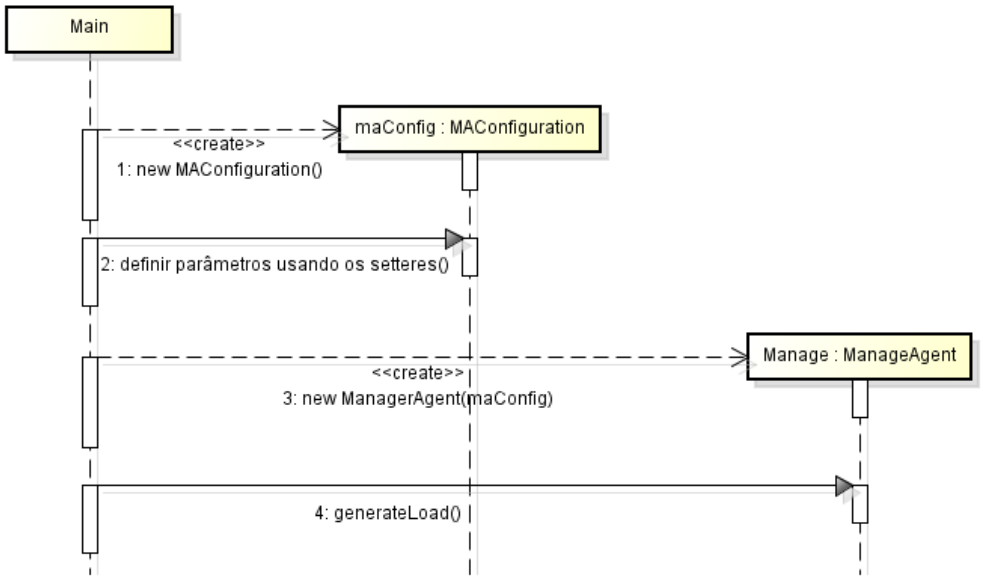


Figura 4.3 Diagrama de sequência principal FlexLoadGenerator

O diagrama de sequência apresentado na Figura 4.3 mostra como ferramentas de geração de eventos devem ser construídas e como as mensagens são trocadas entre objetos. A classe principal do projeto é representada pelo retângulo que contém a palavra '*Main*'. Nesta classe deve ser criado uma instância da classe *MAConfiguration*, referenciada através da variável *maConfig*, conforme mostra a mensagem 1 do diagrama da Figura 4.3. O próximo passo é configurar os parâmetros *setDist()*, *setAmountOfPR()*, *addParameter()*, *setParallel()* e *setRound()*. São estes parâmetros que vão definir como os eventos devem ser criados e gerenciados no objeto *maConfig* (mensagem 2 do diagrama). Uma vez que os parâmetros de geração estiverem adequadamente definidos, uma instância da classe *ManagerAgent* referenciada pela variável *manager* deverá ser criado (mensagem 3 do diagrama); o objeto da classe *MAConfiguration* é passado como parâmetro para o método construtor de *ManagerAgent*. A última mensagem no diagrama (mensagem 4) representa uma chamada para o método *generateLoad()*, que é responsável por criar e

gerenciar os eventos criados.

A maneira como a geração de eventos é executada depende de como os parâmetros *isParallel()* e *isRound()* são definidos no objeto *maConfig* (mensagem 2 da Figura 4.3). As Figura 4.4, Figura 4.5, Figura 4.6 e Figura 4.7 apresentam os diagramas de sequência em conformidade com as diferentes configurações possíveis entre estes dois parâmetros da classe *MAConfiguration*.

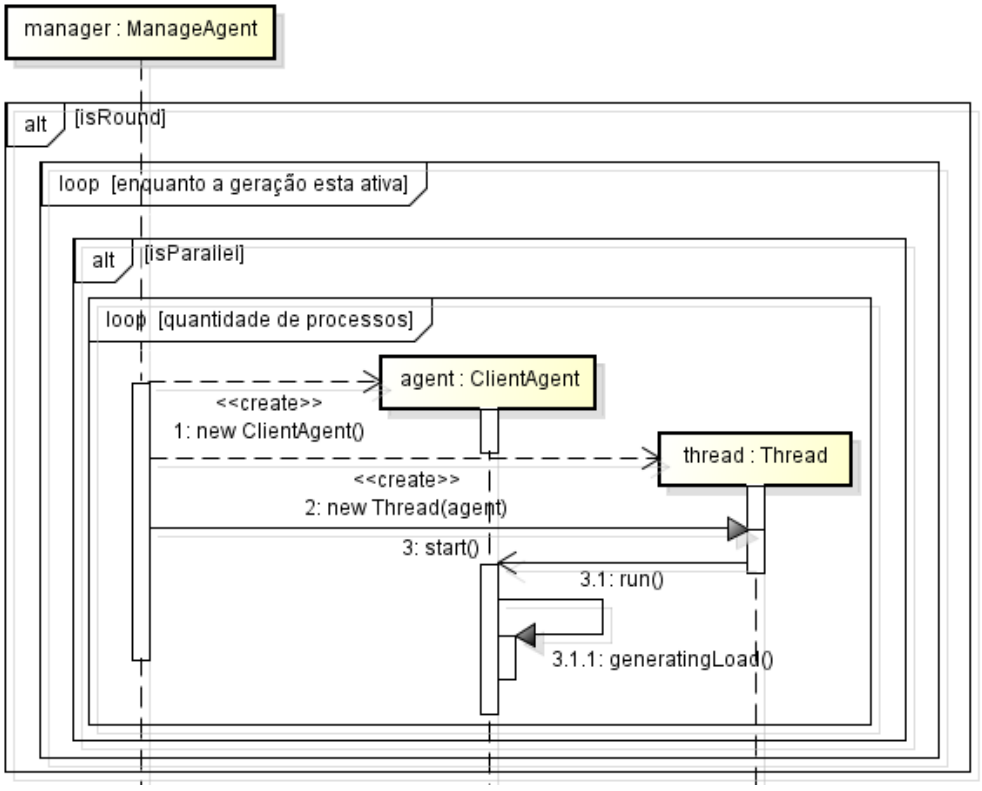


Figura 4.4 Diagrama de sequência para o processo de criação e gerenciamento de eventos a partir da execução de *generateLoad()* quando *setParallel()* e *setRound()* são configurados como *true*

O diagrama da Figura 4.4 representa o processo de geração de carga de trabalho, após a chamada do método *generateLoad()* (mensagem 4 do diagrama de sequência da Figura 4.3), quando os parâmetros *isParallel()* e *isRound()* estão definidos como *true* (retângulos com rótulos *alt*). Neste caso, o processo é realizado em dois laços de repetição (retângulos com rótulos *loop*). O primeiro é utilizado para gerar, em intervalos de tempos aleatórios, as novas instâncias de objetos *ClientAgent*. Os intervalos de tempo são definidos em conformidade com a distribuição de probabilidade escolhida (*setDist()*) e valor(es) informado(s) a esta distribuição (*setParameter()*). O segundo laço é utilizado para gerar o número de instâncias (*setAmountOfPR()*) a serem executadas em paralelo.

O primeiro ciclo do diagrama da Figura 4.4, será implementado durante o processo de geração de eventos. Enquanto o segundo será executado de acordo com o número de processos (clientes) em paralelo previamente configurado.

Após a inicialização dos dois laços, é criada uma instância da classe *ClientAgent* (mensagem 1), que é passada como parâmetro para a criação de uma instância da classe *Thread* (classe disponibilizada por *java.lang.Thread*), presente na mensagem 2 da Figura 4.4 (seta tracejada que contém o rótulo *2: new Thread(agent)*). O método *start()* do objeto *thread* (mensagem 3) chama o método *run()* (mensagem 3.1) do objeto *agent* que por sua vez chama o método *generatingLoad()*(mensagem 3.1.1) inicializando execução em paralelo. Quando o desenvolvedor for descrever o evento para a geração de carga, este deve criar uma classe, estender uma das classes *ClientAgentTCP*, *ClientAgentUDP* ou *ClientAgentSSH*, e sobrescrever o método *generatingLoad()*.

A Figura 4.5 representa a geração de eventos quando os parâmetros *isRound()* e *isParallel()* são configurados como *true* e *false*, respectivamente. Neste caso, o objeto *managerAgent* cria apenas uma instância do agente (mensagem 1), entra em um *loop* e em seguida chama o método *generatingLoad()* do objeto *agent*.

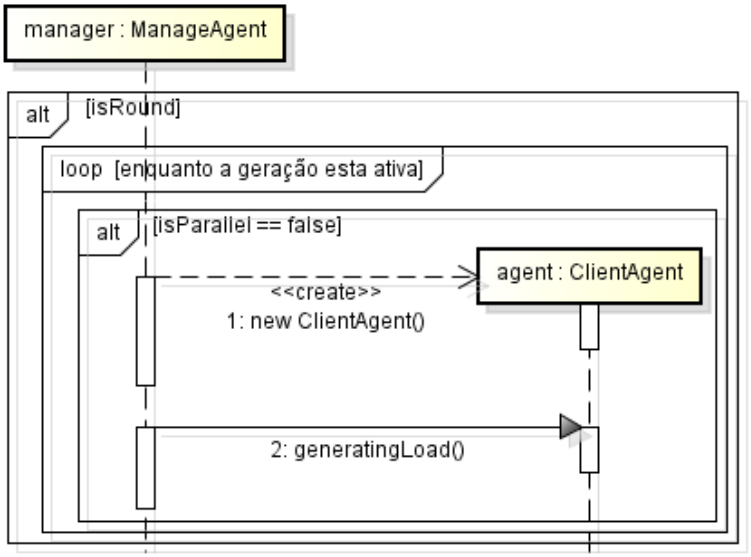


Figura 4.5 Diagrama de sequência para o processo de criação e gerenciamento de eventos a partir da execução de *generateLoad()* quando *setParallel()* é configurado como *false* e *setRound()* é configurado como *true*

O processo apresentado na Figura 4.5 difere do apresentado na Figura 4.3, devido a geração ser efetuada apenas de forma sequencial já que, neste caso, o parâmetro *isParallel()* é ajustado como *false*. O método *generatingLoad()* é chamado diretamente pelo *managerAgent* (mensagem 2) em intervalos de tempo distribuídos aleatoriamente,

de acordo com a distribuição e parâmetros informados em *setDist()* e *addParameter()* (mensagem 2 da Figura 4.3).

O diagrama da Figura 4.6 representa a geração de eventos quando os parâmetros *isRound()* e *isParallel()* são configurados como *false* e *true*, respectivamente. Neste caso, os eventos são gerados em paralelo, porém, existe apenas uma rodada de geração. Desta forma, o objeto *managerAgent* cria uma instancia da classe *ClientAgent*, entra em um *loop*, e cria a quantidade de instância em paralelo definido pelo parâmetro, *setAmountOfPR()* (mensagem 2 da Figura 4.3). Como as rodadas não são gerenciadas pelo *ManagerAgent*, devem ser implementadas pelo desenvolvedor na criação do *Agent*.

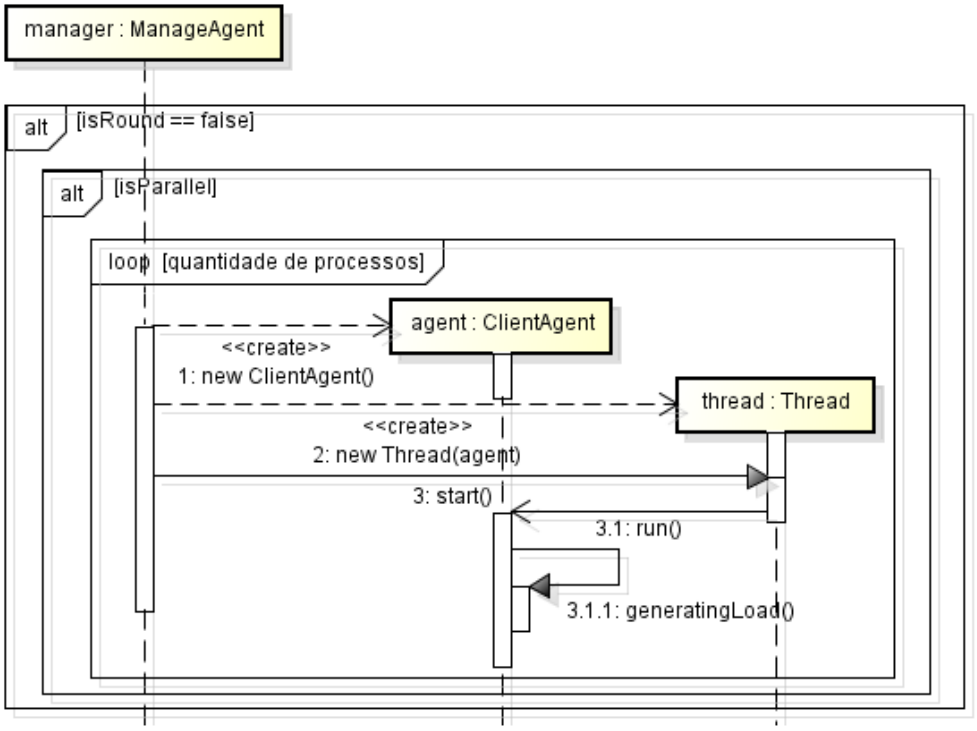


Figura 4.6 Diagrama de sequência para o processo de criação e gerenciamento de eventos a partir da execução de *generateLoad()* quando *setParallel()* é configurado como *true* e *setRound()* é configurado como *false*

O diagrama da Figura 4.7 mostra o processo de execução quando os métodos *setRound()* e *setParallel()* são definidos como *false*. Neste caso somente uma instância de *ClientAgent* (mensagem 1) é gerada e o método *generatingLoad()* (mensagem 2) é chamado apenas uma vez de modo que o desenvolvedor é responsável pela implementação das rodadas que permitirão a continuidade da geração dos eventos.

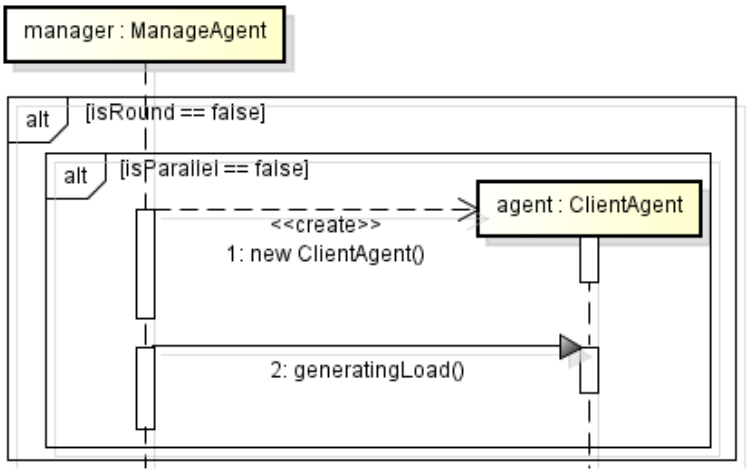


Figura 4.7 Diagrama de sequência para o processo de criação e gerenciamento de eventos a partir da execução de *generateLoad()* quando *setParallel()* e *setRound()* são configurados como *false*

4.4 Composição FlexLoadGenerator

O FlexLoadGenerator é composto por onze classes no total, e uma biblioteca responsável por geração de números aleatórios. Para facilitar a compreensão da descrição das classes e métodos presentes nesta seção, o *framework* foi dividido logicamente em duas partes: (1) classes de comunicação e (2) classes de criação e gerenciamento de eventos. A primeira parte é responsável por estabelecer a comunicação entre o ferramental desenvolvido e o sistema alvo, enquanto, a segunda, se encarrega da criação e controle de eventos. Entretanto, as classes *AgentClassNotProvidedException* e *WrongAmountOfParametersException* não foram incluídas nesta divisão lógica por serem utilizadas para o tratamento de exceções internas quanto ao uso indevido do *framework*. A biblioteca de geração de números aleatórios também não integra a divisão lógica anteriormente estabelecida.

As subseções a seguir discutem a biblioteca de geração de números aleatórios, e as classes que compõem o *framework*, com exceção das classes *AgentClassNotProvidedException* e *WrongAmountOfParametersException* que já tiveram seus propósitos descritos.

4.4.1 O Núcleo de Geração de Números Aleatórios

A biblioteca de geração de números aleatórios é responsável por auxiliar no gerenciamento entre a ocorrência de disparos de eventos. A biblioteca gera amostras com

valores aleatórios baseados em algumas das principais distribuições de probabilidade existentes (contínuas e discretas), através da aplicação de técnicas de geração de variáveis aleatórias (*random variates*) (Devroye, 1986). As distribuições de probabilidade mais amplamente utilizadas nas áreas de avaliação de desempenho e dependabilidade de sistemas computacionais (Jain, 1991; Hayter, 2007) estão implementadas nesta biblioteca, são elas: Erlang, Exponencial, Log-normal, Normal, Pareto, Triangular, Weibull e Uniforme (contínuas), e Geométrica e Poisson (discretas). Além das distribuições mencionadas esta biblioteca também possui a distribuição Empírica implementada.

Esta biblioteca advém do *kernel* presente no WGCap (*Workload Generator for Capacity Advisor*), ferramenta concebida por Galindo *et al.* (2009), sem sofrer qualquer alteração em sua codificação. Para tanto, apenas foram preservados os pacotes, e suas respectivas classes, referentes a instruções para funcionamento de cada distribuição de probabilidade, controle, e entrada e saída de informações. O pacote responsável pela GUI (*Graphical User Interface*) foi descartado por não ser necessário. A geração de números aleatórios através das funções de distribuições de probabilidade promovida pelo *kernel* do WGCap foi amplamente testada e sua eficácia devidamente comprovada, como exposto em (Galindo *et al.*, 2010).

O uso das distribuições anteriormente mencionadas neste trabalho é feito por meio de constantes, onde a nomenclatura de cada constante é o nome de distribuição no formato caixa alta, em inglês. Por exemplo, exponencial corresponde a EXPONENTIAL. A única exceção corresponde a distribuição Empírica. Para fazer uso desta distribuição é preciso acessar diretamente o pacote *com.gcap.randomvariategenerator.basics*. Optou-se por manter esta distribuição por ser útil quando se deseja obter valores sintéticos próximos aos reais de algum sistema, quando estes não correspondem às demais distribuições disponíveis no *framework*.

4.4.2 Classes de Comunicação

As classes que integram esta parcela da divisão lógica são: *ClientAgent*, *ClientAgentTCP*, *ClientAgentUDP*, *ClientAgentSSH*, *ClientFunctionTCP*, *ClientFunctionUDP* e *ClientFunctionSSH*. Estas classes são responsáveis por estabelecer a comunicação com o sistema em teste, desde que este se comunique através do protocolo TCP (*Transfer Control Protocol*), UDP (*User Datagram Protocol*) ou SSH2 (*Secure Shell 2*).

A classe *ClientAgent* é a uma classe do tipo abstrata, superclasse de *ClientAgentTCP*, *ClientAgentUDP* e *ClientAgentSSH* portanto, as três classes herdam os métodos abstratos presente na classe pai. O método *generatingLoad()* oferecido pela classe *ClientAgent*

é o principal ponto de *hotspot* do *framework*. É através da sobrescrita deste método que o desenvolvedor adiciona seu código definindo que evento será criado. Este método é acessado apenas a partir da extensão de uma das subclasses de *ClientAgent* (classes filhas). É importante frisar que a subclasse a ser estendida dependerá do protocolo que se deseja utilizar para estabelecer a comunicação.

A comunicação propriamente dita é feita por uma das classes *ClientFunctionTCP*, *ClientFunctionUDP* ou *ClientFunctionSSH*. O usuário não necessita chamar nenhuma destas classes, uma vez que cada uma delas está associada ao seu *ClientAgent* respectivo, por exemplo, *ClientAgentTCP* esta diretamente associada a *ClientFunctionTCP*.

A lista completa dos construtores e métodos pertencentes as classes destinadas a comunicação pode ser visto no Apêndice B.

4.4.3 Classes de Criação e Gerenciamento de Eventos

As classes que fazem parte desta parcela da divisão lógica do FlexLoadGenerator são: *MAConfiguration()* e *ManagerAgent()*.

Em *MAConfiguration()* é definido como os eventos deverão ser criados (sequencial ou paralelo) e como estes devem interagir com o sistema em teste. O comportamento da ferramenta construída com o auxílio do *framework* dependerá da forma de configuração dos métodos desta classe. Dois dos mais importantes métodos são: *setParallel(boolean isParallel)* e *setRound(boolean isRound)*, ambos de caráter booleano. Estes métodos determinam se a geração de eventos ocorrerá em paralelo (*setParallel(boolean isParallel)*) e como os processos (eventos) devem ser mantidos (*setRound(boolean isRound)*). As quatro possíveis combinações desses métodos podem ser vistos nas Figuras 4.4 (*setParallel(boolean isParallel)* e *setRound (boolean isRound)* são definidos como verdadeiros), 4.5 (apenas *setRound(boolean isRound)* definido com verdadeiro), 4.6 (apenas *setParallel(boolean isParallel)* definido com verdadeiro) e 4.7 (*setParallel(boolean isParallel)* e *setRound(boolean isRound)* são definidos como falsos). Os demais métodos desta classe determinam a quantidade de processos (eventos), a distribuição de probabilidade (juntamente com seus parâmetros) para geração de intervalos de tempos entre eventos, dentre outros. É importante observar que esta classe é a responsável por controlar os eventos de acordo com a configuração que o objeto desta classe receber.

A classe *ManagerAgent()* se encarrega de efetivamente criar os eventos de acordo com o que foi definido em *MAConfiguration()*.

Maiores detalhes sobre estas classes e seus respectivos construtores e métodos podem ser encontradas no Apêndice B.

4.5 Exemplo da Construção de Geradores de Eventos

Nesta seção é apresentado um exemplo de gerador de eventos construído com o auxílio do FlexLoadGenerator. A aplicação que serve de exemplo, denominada Kangaroo, realiza unicamente solicitações à página principal de **www.modcs.org**, presente na Web.

O Kangaroo é composto por duas classes, onde a primeira possui o mesmo nome da aplicação e a segunda foi nomeada *Agent*. O objetivo da classe Kangaroo é definir como a aplicação deve se comportar ao longo de sua execução. Devido isto, é nesta classe que os parâmetros pertencentes a classe *MAConfiguration* são configurados. A classe Kangaroo pode ser vista logo a baixo.

```
1  package kangaroo;
2
3  import com.generator.*;
4  import java.lang.reflect.InvocationTargetException;
5  import java.util.logging.Level;
6  import java.util.logging.Logger;
7
8  /**
9   * Classe principal do projeto Kangaroo
10  */
11 public class Kangaroo {
12
13     public static String ip = "www.modcs.org";  // Ip do sistema em
           teste
14     public static String resource = ""; //Especifica o recurso o
           solicitado. Neste caso, por estar vazio, significa a chamada
           da pagina principal
15
16     public static void main(String[] args) {
17
18         MAConfiguration ma = new MAConfiguration();
19
20         ma.setIpAddress(ip);
21         ma.setPort(80);
22         ma.setAgent(Agent.class); // Recebe a classe contendo o evento
               a ser gerado.
23         ma.setDist(ManagerAgent.EXPONENTIAL);
24         ma.setAmountOfPR(5);
25         ma.addParameter(500.0);
26         ma.setParallel(true);
```

```
27          ma.setRound(true);
28
29          ManagerAgent managerAgent = null;
30
31          //Recebe o objeto ma da classe MAConfiguration para fazer o
                gerenciamento da ferramenta
32          managerAgent = new ManagerAgent(ma);
33
34          try {
35              managerAgent.generateLoad(); //Realiza o gerenciamento dos
                    eventos.
36          } catch (IllegalArgumentException ex) {
37              Logger.getLogger(Kangaroo.class.getName()).log(Level.
                    SEVERE, null, ex);
38          } catch (SecurityException ex) {
39              Logger.getLogger(Kangaroo.class.getName()).log(Level.
                    SEVERE, null, ex);
40          } catch (InstantiationException ex) {
41              Logger.getLogger(Kangaroo.class.getName()).log(Level.
                    SEVERE, null, ex);
42          } catch (IllegalAccessException ex) {
43              Logger.getLogger(Kangaroo.class.getName()).log(Level.
                    SEVERE, null, ex);
44          } catch (InvocationTargetException ex) {
45              Logger.getLogger(Kangaroo.class.getName()).log(Level.
                    SEVERE, null, ex);
46          }
47      }
48  }
```

Listing 4.1 Classe Kangaroo.

Conforme pode ser observado no código da classe Kangaroo, a aplicação assume o comportamento descrito no diagrama de sequência da Figura 4.4, pois o objeto (*ma*) de *MAConfiguration* foi definido como paralelo e em rodadas (*setParallel(true)* e *setRound(true)*). Isto faz com que a aplicação fique requisitando 5 vezes (*setAmountOfPR(5)*) a página armazenada no servidor endereçado pela variável *ip*, que por sua vez é atribuída como parâmetro à *setIpAddress(ip)*, e na porta especificada (*setPort()*), sem necessariamente aguardar o retorno dado pelo servidor para fazer novas solicitações. O tempo entre requisições é definido através de *setDist(ManagerAgent.EXPONENTIAL)* e *addParameter(500)*, onde o primeiro trata da escolha da distribuição e o segundo o valor informado à distribuição para geração dos intervalos de tempos em milissegundos

(500). O parâmetro *setAgent(Agent.class)* recebe a classe que contém o evento a ser gerado. Esta classe é descrita na subseção 4.5.1.

Para dar início ao funcionamento da ferramenta o objeto da classe *ManagerAgent* (*managerAgent*) recebe como parâmetro o objeto da classe *MAConfiguration* (*ma*) devidamente configurado. É importante observar que o Kangaroo é uma aplicação de exemplo, e devido à sua simplicidade, o Kangaroo ficará funcionando indefinidamente após a chamada do método *generatingLoad()* de *ManagerAgent*, pois não foi estabelecido qualquer período de tempo para execução. Em consequência disto a aplicação somente poderá ser encerrada manualmente pelo usuário.

4.5.1 A classe *Agent*

A classe *Agent* é responsável por definir o evento que deverá ser criado e gerenciado pela classe Kangaroo. É nesta classe que o desenvolvedor deve informar qual protocolo de comunicação foi escolhido (TCP, UDP ou SSH2). No exemplo optamos por estabelecer a comunicação com o protocolo TCP através da extensão (*extends*) da classe *ClientAgentTCP*, pois o protocolo HTTP utiliza o TCP como protocolo de transporte. A partir da extensão desta classe o *framework* solicita que o método *generatingLoad()* seja incluído. É no corpo deste método que o desenvolvedor deve especificar o evento a ser criado. Como o Kangaroo solicita a página principal de **www.modcs.org** então no interior do método *generatingLoad()* apenas são enviadas mensagens solicitando a página identificada na variável *resource* que está na classe Kangaroo. O código da classe *Agent* pode ser vista a seguir.

```
1  package kangaroo;
2
3  import com.generator.*;
4  import com.generator.ClientAgentTCP;
5  import java.io.IOException;
6  import java.util.logging.Level;
7  import java.util.logging.Logger;
8
9  /**
10 * Classe encarregada de definir os eventos a serem criados
11 */
12 public class Agent extends ClientAgentTCP{
13
14     public Agent() throws IOException{
15     }
```

```
16
17      @Override
18      // Sobrescrita do metodo generatingLoad()
19      public void generatingLoad() {
20          try {
21              //Mensagem enviada ao servidor solicitando recurso
22              String s = "GET /"+Kangaroo.resource+" HTTP/1.1\r\n"
23                      + "Host:"+Kangaroo.ip+"\r\n\r\n";
24
25              //Estabelecimento do tamanho do buffer
26              getClientFunction().setBufferSizer(2048);
27
28              //Envia a mensagem e escreve a resposta do servidor
29              getClientFunction().sendMessage(s);
30
31          } catch (IOException ex) {
32              Logger.getLogger(Agent.class.getName()).log(Level.SEVERE,
                    null, ex);
33          }
34      }
35  }
```

Listing 4.2 Classe Agent.

Depois de finalizado a codificação da classe *Agent* esta é passada como parâmetro *setAgent(Agent.class)* para o objeto *ma* da classe Kangaroo para criação do evento.

4.6 Avaliação dos métodos presente no FlexLoadGenerator

Visando garantir a eficiência do FlexLoadGenerator, todos os métodos que o constituem foram devidamente checados. Para testar os métodos e verificar se correspondem ao desejado, inicialmente, estes foram divididos em grupos. Os grupos foram estabelecidos de acordo com o protocolo de comunicação, totalizando assim três grupos (TCP, UDP e SSH2). No grupo TCP houve uma subdivisão devido a impossibilidade de utilizar os métodos sendMessage(String msg) e *justSendMessage(String msg)* ao mesmo tempo, visto que ambos têm a mesma tarefa de enviar mensagens para o sistema em teste, com exceção de *sendMessage(String msg)*, pois este também retorna uma mensagem. Os métodos das classes *MAConfiguration* e *ManagerAgent* foram repetidamente testados por estarem inclusas nos três grupos.

O teste aplicado aos grupos de métodos TCP, UDP e SSH2 contou com a modificação do código do Kangaroo, exemplo de ferramenta concebida com o auxílio do *framework*, apresentado na seção 4.5, alterando os protocolos de comunicação e inserindo os métodos pertencentes a cada grupo, respeitando as divisões estabelecidas.

Para realização dos testes, três pequenos servidores foram construídos, um para cada protocolo de comunicação. Em cada servidor aplicou-se mecanismos que poderiam confirmar, por exemplo, se uma comunicação foi estabelecida com sucesso.

O teste aos quais os métodos foram submetidos objetivaram comparar os resultados obtidos com o resultado esperado. Por exemplo, o método *isParallel()* tem por função retornar verdadeiro ou falso, já que é do tipo booleano e se destina a informar se a geração irá ocorrer de forma paralela (múltiplos processos, também chamado de eventos, executando ao mesmo tempo) ou não.

A avaliação dos métodos contou apenas com uma única máquina que abrigou tanto as variações do Kangaroo (para TCP, UDP e SSH2) como os servidores, com quem se deveria estabelecer a comunicação. O resultado dos testes se encontra na Tabela 4.1. No entanto, devido a grande quantidade de métodos presente no *framework*, a Tabela 4.1 apenas contém os resultados dos testes de métodos que não foram utilizados na construção de ferramentas geradoras de eventos (forma de validação do *framework* adotada neste trabalho discutida no capítulo 5).

Tabela 4.1: Resultados dos testes de alguns dos métodos disponibilizados pelo FlexLoadGenerator

Grupo	**Classe**	**Método**	**Resultado esperado**	**Resultado obtido**
1	ClientFunctionTCP	sendMessage(String msg)	Cabeçalho e página em HTML	Cabeçalho e página em HTML
1.1	ClientFunctionTCP	justSendMessage(String msg)	Confirmação de conexão com o servidor teste	Conexão estabelecida com sucesso
1	ClientFunctionTCP	getRequestTime()	Tempo, em ms, decorrente do último envio de mensagem	2ms

1	ClientFunctionTCP	getResponseByteLenght()	Quantidade de bytes da última resposta enviada pelo servidor	7 (sucesso)
2	ClientFunctionUDP	justSendMessage(String msg)	Confirmação de conexão com o servidor teste	Conexão estabelecida com sucesso
3	ClientFunctionSSH	sendMessage(String msg)	Mensagem de retorno do comando enviado	sim
3	ClientFunctionSSH	getRequestTime()	Tempo, em ms, decorrente do último envio de mensagem	10ms
3	ClientFunctionSSH	getResponseByteLenght()	Quantidade de bytes da última resposta enviada pelo servidor	411 (retorno da execução do comando listar)
*[1]	MAConfiguration	getDist()	Número inteiro correspondente a distribuição escolhida	2 (Exponencial)
*[1]	MAConfiguration	getListParameters()	Lista de parâmetros informados para distribuição	[0.0, 1.0E7, 135.0]
*[1]	MAConfiguration	getAmountOfPR()	Quantidade de eventos	10
*[1]	MAConfiguration	isRound()	Interatividade na criação de eventos?	Sim
*[1]	MAConfiguration	isParallel()	Eventos em paralelo?	Sim

Como é possível observar na Tabela 4.1 os métodos pertencentes ao *framework* provê corretamente as funcionalidades para os quais foram desenvolvidos. É importante frisar alguns pontos dos resultados apresentados. O primeiro deles se encontra na terceira linha da referida tabela, onde esta escrito "'2ms". Este resultado, com período de tempo tão curto, se deve ao Kangaroo e o servidor TCP residirem na mesma máquina. Esta mesma observação pode ser feita a linha 7, onde se lê "10ms". O resultado apresentado na linha 4 retrata a quantidade de *bytes* da última resposta dada pelo servidor, neste caso o servidor havia sido informado a responder a palavra "sucesso" caso a comunicação fosse estabelecida com sucesso. Uma observação deve ser feita em relação aos resultados presentes nas linhas 6 e 8. Devido ao protocolo de teste dessas linhas ser o SSH2 é essencial informar que o servidor, neste caso, foi instruído a responder "sim" para comunicação estabelecida com sucesso. Já em relação a oitava linha, o resultado de 411 *bytes* se deve ao retorno da execução do comando **ls** (listar o que estiver na pasta ou diretório), onde, a pasta de em questão continha apenas um arquivo.

Um outro ponto que merece destaque são as linhas da Tabela 4.1 sinalizadas com "*". Os métodos presentes nestas linhas foram testados em todos os três grupos, mas os resultados apresentados na tabela são referentes ao grupo 3 (SSH2). Na nona linha a distribuição escolhida para execução dessa variação do Kangaroo foi a exponencial, que, neste caso, corresponde a constante EXPONENTIAL definida como número 2. O resultado da linha 10 mostra a lista de valores informados para distribuição escolhida. Os dois primeiros valores são os intervalos de números (o menor e o maior valor) que a distribuição pode gerar, este intervalo numérico é um padrão do *framework* imposto pela a biblioteca de geração de números aleatórios, o terceiro número (135.0) é o tempo que foi informado para o intervalo da ocorrência de eventos, em milissegundos. As linhas 13 e 11 reportam apenas que 10 processos (eventos) deveriam ser disparados simultaneamente. E, por fim, o resultado presente linha 12 mostra que estes eventos deveriam ser novamente acionados de tempos em tempos determinado pela distribuição e seus parâmetros.

4.7 Considerações Finais

Este capítulo apresentou o FlexLoadGenerator e os aspectos que motivaram o seu desenvolvimento, além da metodologia adotada para sua concepção. A documentação referente aos modelos em UML foi apresentada. Em seguida, a composição do *fra-*

[1]Método testado nos três grupos.

mework foi exposto juntamente com o núcleo de geração de números aleatórios, seguido pelos meios de comunicação, criação e gerenciamento de eventos. Um exemplo prático de utilização do *framework* foi exposto de modo a demonstrar sua aplicabilidade. Por fim, foi abordado a avaliação dos métodos que compõem o *framework* demonstrando que este é capaz de oferecer corretamente suas funcionalidades.

5

Visão Geral dos Ferramentais Desenvolvidos Tendo por Base o FlexLoadGenerator

Para se ter sucesso, é necessário amar de verdade o que se faz.

—STEVE JOBS

Este capítulo apresenta a validação do FlexLoadGenerator analisando sua efetividade através da concepção de duas ferramentas geradoras de eventos, onde a primeira ferramenta é destinada a auxiliar em estudos de avaliação de desempenho enquanto a segunda, é direcionada para testes de dependabilidade, mais especificamente confiabilidade e disponibilidade. O principal objetivo deste capítulo é observar a redução no esforço necessário para codificação destas ferramentas.

Neste capítulo primeiramente é apresentado uma breve introdução sobre pagamento eletrônico, seguido do sistema de transferência eletrônica de fundos (TEF), visto que esta é a área de atuação da primeira ferramenta. Na sequência, a ferramenta WGSysEFT é apresentada juntamente com suas características e forma de desenvolvimento. Posteriormente, computação em nuvem é contextualizado, seguido pela plataforma Eucalyptus (área de atuação da segunda ferramenta). Por fim, o ferramenta EucaBomber é apresentada, descrevendo suas principais características e aspectos de desenvolvimento.

5.1 WGSysEFT

Esta seção é dedicada a contextualizar pagamento eletrônico e o sistema de transferência eletrônica de fundos (TEF), apresentando a motivação de se construir um gerador

de carga de trabalho sintética que atue junto a este sistema para propiciar a criação de experimentos para avaliação de desempenho. Em seguida, a ferramenta proposta é descrita e, por fim, é apresentado a forma de implementação desta ferramenta, em detalhes.

5.1.1 Pagamento Eletrônico

Grande parte dos estabelecimentos comercias presentes no Brasil oferecem aos seus clientes formas de pagamentos por meio eletrônico. Esta comodidade permite ao consumidor adquirir bens ou serviços, mesmo que no momento da compra esteja desprovido de qualquer quantia em dinheiro. A quitação de débito por meio eletrônico permite que o cliente realize compras quando desejar, de forma prática e segura. Enquanto que, para o comerciante representa aumento nas vendas e garantia de ressarcimento do valor das mercadorias comercializadas.

O pagamento eletrônico é qualquer tipo de pagamento que não faz uso de papel moeda, cheque ou qualquer outro tipo de pagamento não eletrônico. Algumas formas de pagamento eletrônico são realizadas por meio de cartões magnéticos, transferências de valores, cartões de vale-alimentação, entre outros.

Compras efetuadas com cartão de crédito possibilitam ao cliente optar por pagamentos à vista ou parcelados. No pagamento à vista, o consumidor quita o débito no momento da compra, mesmo que este não possua o valor do bem ou serviço no momento da aquisição. Essa modalidade permite que o estabelecimento comercial seja pago na data estipulada entre este e a instituição mantenedora do cartão. Enquanto que a instituição mantenedora do cartão aguarda o ressarcimento do valor pago à instituições comerciais por meio de cobranças enviadas aos clientes (ABECS, 2013).

Na compra parcelada, o valor do bem ou serviço é dividido em parcelas, onde a quantidade de parcelas é estabelecida entre o consumidor e o comerciante no ato da aquisição. As prestações são cobradas mensalmente aos clientes por meio de faturas.

A abertura de contas em instituições bancárias possibilita a emissão de cartões de débito. Isto, devido a não necessidade de aprovação de crédito, ou a cobrança de qualquer tarifa. O pagamento de importâncias com cartão de débito ocorre de maneira bastante simples. Quando o cliente opta em realizar o pagamento por meio de cartão de débito, a quantia informada é debitada diretamente da conta do cliente (ABECS, 2013), e repassado ao estabelecimento comercial. Ainda em relação a compras com cartão de débito, é preciso mencionar que outra modalidade deste tipo de pagamento é realizada, o débito pré-datado.

Embora pouco popular no Brasil, pagamentos também podem ser efetuados através

de cartões de débito pré-datado (Cielo, 2013; Bradesco, 2013). Esta modalidade funciona de maneira semelhante ao cheque de papel. Datas de pagamentos e valores a serem descontados são acordados entre o cliente e o estabelecimento comercial. Na data estabelecida, a instituição financeira retira o valor estabelecido da conta do cliente. Caso o cliente não possua a importância estipulada em sua conta na data programada para retirada, a instituição financeira efetua o pagamento para o estabelecimento comercial e inicia um processo de cobrança junto ao cliente (Cielo, 2013).

Algumas instituições financeiras (Bradesco, 2013; Itaú, 2013) já aderiram a essa forma de pagamento. Porém, nem todos os estabelecimentos comerciais aceitam esse tipo de quitação de débito.

5.1.2 O Sistema de Transferência Eletrônica de Fundos

O sistema de TEF é uma solução de pagamento eletrônico através do qual os clientes podem efetuar pagamentos a estabelecimentos comerciais. A ordem de autorização para quitação de débito ocorre por meio de empresas autorizadoras. Estas empresas avaliam dados, tais como saldo, lista negra, senha, validade do cartão, limite de compras, dentre outros. Caso a empresa autorizadora verifique que tais condições estão em ordem, o pagamento é realizado.

Uma solução TEF lida com três elementos: PDV (Ponto de Venda), servidor TEF, e as redes autorizadas (Itautec, 2013a). Estes elementos são descritos como segue:

- **PDV:** geralmente é um dispositivo móvel, onde cartões podem ser lidos (tarja magnética) ou inseridos (*chip*) para realização de transações. É no PDV que a transação é efetivamente articulada. O *software* interno ao PDV é responsável pela montagem e troca de mensagens entre o dispositivo e o servidor TEF;
- **Servidor TEF:** máquina onde é instalado o gerenciador de transações TEF. Este gerenciador é responsável por processar e armazenar transações, além de estabelecer comunicação com redes autorizadoras;
- **Redes autorizadoras:** são organizações que gerenciam as operações feitas a partir de cartões de crédito e débito. Também são responsáveis pela comunicação entre o estabelecimento comercial onde o bem, ou serviço, foi adquirido, e o banco do cliente.

Na mesma máquina onde o gerenciador de transações é instalado, pode-se incluir também o banco de dados responsável por registrar o montante de transações realizadas

no estabelecimento comercial. Porém, não necessariamente, o banco de dados precisa estar no mesmo servidor que o gerenciador de TEF. Itautec (2013a) mostra que uma solução TEF pode ter sua arquitetura descentralizada. Esta opção de instalação é aconselhável caso a empresa possua um tráfego intenso de transações, ou ainda, quando possuem diversas filiais e uma grande quantidade de PDVs. A quantidade de PDVs disponibilizados para uso deve ser cadastrado junto ao servidor TEF.

As mensagens trocadas entre as partes de uma solução de TEF são geralmente codificadas, e seguem o padrão ISO 8583 (ISO, 2013). A Figura 5.1 apresenta as trocas de mensagem entre os componentes do sistema TEF, em detalhes, para a realização de uma transação completa e bem sucedida, descrito por (Itautec, 2013b).

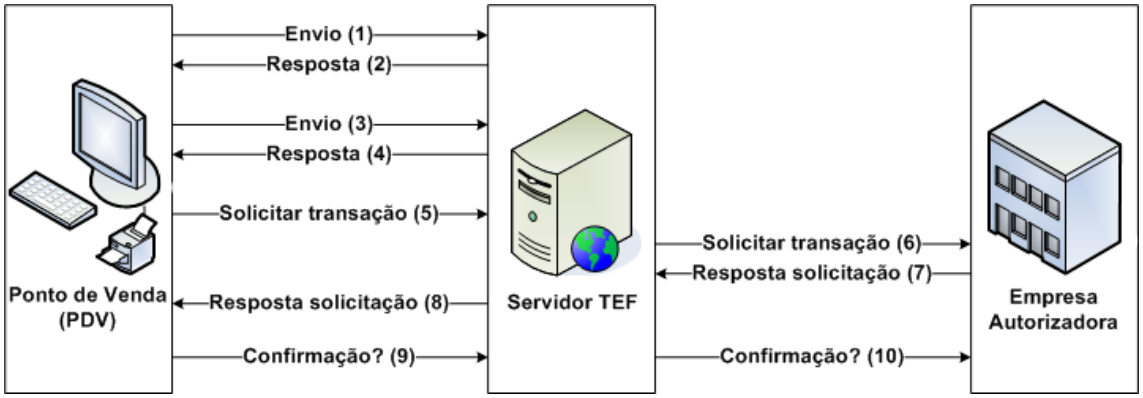


Figura 5.1 Fluxo de mensagens do sistema TEF

No processo mostrado na Figura 5.1, a mensagem de solicitação de transação é enviada duas vezes (fluxo 1 e 3), mas com diferentes objetivos. A primeira mensagem (fluxo 1) é enviada solicitando uma transação. Esta mensagem inclui alguns dados, como o BIN (Número de Identificação Bancária) e um grupo de serviços. Algumas outras informações são transmitidas ainda na primeira mensagem, de acordo com o estabelecido na norma ISO 8583. A segunda mensagem é destinada a produtos selecionados (fluxo 3), como por exemplo, pagamento com cartão de débito à vista (REDECARD, 2013). A resposta do servidor TEF dependerá do serviço solicitado. A primeira resposta (fluxo 2) é composta por rede autorizadora, bandeira, serviços disponíveis, além de atributos como limite de parcelas, data limite de agendamento, entre outros. Enquanto na resposta para a segunda solicitação (fluxo 4), o servidor de TEF informa como as mensagens seguintes devem ser montadas para envio. Os primeiros dois pares de mensagens trocadas não envolvem a comunicação com a rede autorizadora. Com a resposta obtida por meio do fluxo 4, o PDV envia uma mensagem de solicitação de transação (fluxo 5). O servidor TEF processa e registra a transação e envia uma mensagem para a empresa autorizadora

responsável pelo cartão de cliente (fluxo 6). Quando a rede autorizadora recebe a mensagem, esta responde ao Servidor TEF informando se a transação foi aprovada ou não (fluxo 7). O servidor TEF encaminha o status de aprovação ou rejeição enviado pela empresa autorizadora ao PDV (fluxo 8). Caso o status da mensagem contenha aprovação da compra, fica então a cargo do cliente a aceitação ou desistência da transação. A resposta do cliente é enviada ao servidor TEF (fluxo 9), que encaminha para a rede autorizadora, concluindo, assim, a transação (fluxo 10).

5.1.3 O WGSysEFT - Visão Geral

O WGSysEFT (*Workload Generator for Electronic Funds Transfer System*) é um gerador de carga de trabalho probabilístico que realiza transações de crédito e débito, a vista ou parcelado, em sistemas TEF. Desenvolvido a partir do FlexLoadGenerator, a ferramenta pode ser utilizada para geração de carga de trabalho para avaliação de desempenho de sistemas TEF.

O WGSysEFT gerencia diversos PDVs para realização de transações. A Figura 5.2 faz um contraste entre a arquitetura básica do sistema TEF (fluxo 1, componentes envolvidos por linha tracejada em vermelho) e a substituição de parte da arquitetura para inclusão da ferramenta de geração de carga sintética proposta (fluxo 2, componentes envolvidos por uma linha tracejada em azul).

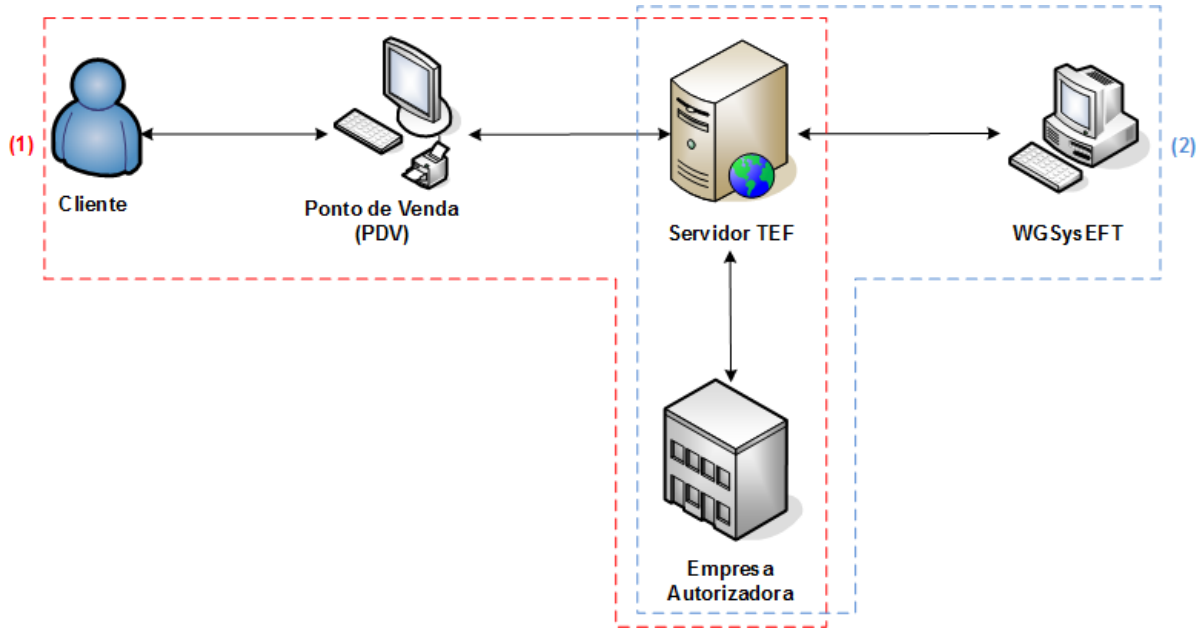


Figura 5.2 Arquitetura básica do sistema TEF com inclusão do WGSysEFT

O fluxo 1 presente na Figura 5.2, mostra o curso habitual para a realização de uma transação. Neste percurso, o PDV recolhe informações cedidas pelo cliente, tais como número do cartão, senha, forma de pagamento, entre outras. Enquanto o PDV estabelece comunicação com o servidor TEF e este, por sua vez, com a empresa autorizadora. Como já descrito anteriormente, para que uma compra seja concluída, os componentes do sistema trocam diversas mensagens até chegar ao final, com a aprovação, ou desistência do cliente.

A substituição das partes que constituem o cliente e o PDV respectivamente, fluxo 1 da Figura 5.2, é feito pelo WGSysEFT (fluxo 2). O WGSysEFT trabalha como cliente, eliminando assim a necessidade de, a cada transação, o PDV ser alimentado com novos dados advindos de algum cliente, e também como PDV enviando e recebendo informações ao servidor TEF.

Embora, a versão atual, realize apenas transações de crédito e débito de forma a vista ou parcelada, pode-se incorporar ao WGSysEFT novas funcionalidades como, por exemplo, inserção de crédito para telefones móveis, compras por meio de cartões de Vale Refeição, entre outras. O manual do usuário contendo maiores informações sobre como utilizar o WGSysEFT pode ser visto no Apêndice C.

5.1.4 Desenvolvimento do WGSysEFT

O projeto de construção do WGSysEFT foi bastante peculiar devido as muitas limitações impostas pelo sistema TEF. Primeiramente, neste projeto não houve qualquer codificação realizada para estabelecer comunicação com o sistema, muito menos se pode aproveitar as formas de comunicações oferecidas pelo *framework*. A comunicação com o sistema somente pode ser estabelecida através de uma biblioteca específica para este fim que é cedida pela empresa que administra o sistema. A segunda particularidade é a não autorização do uso de *threads* atuando como PDVs para solicitações de transações simultâneas ao sistema. Esta última afirmação obrigou-nos a optar pela forma de geração de eventos exposta no diagrama de sequência da Figura 4.7, presente no capítulo 4.

Para contornar a limitação quanto ao uso de *threads* optou-se por construir a aplicação em dois projetos distintos. O primeiro projeto contém apenas as classes de *interfaces* gráfica e uma classe que armazena temporariamente as opções escolhidas pelo usuário no preenchimento das *interfaces* gráficas. O segundo projeto é a ponte que liga o primeiro projeto ao *framework*. É no segundo projeto que reside a classe que determina que eventos devem ser criados.

O desenvolvimento do WGSysEFT aproveita do *framework* todas as 10 distribuições de probabilidades oferecidas (ver a relação das distribuições no capítulo 4, subseção 4.4.1), assim como o meio para criação de eventos e gestão da ferramenta. Esse aproveitamento reduz a quantidade de código que seria necessário desenvolver para que a ferramenta oferecesse as mesmas funcionalidades. Os pacotes e classe que integram o primeiro projeto que compõe o WGSysEFT é apresentado na Figura 5.3.

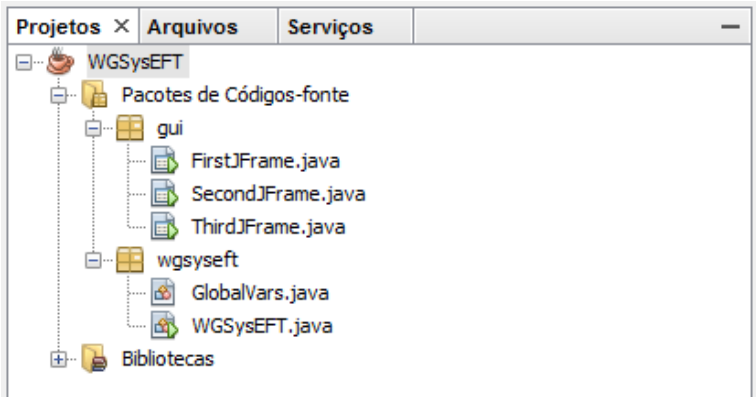


Figura 5.3 *Screenshot* - primeira parte do projeto WGSysEFT

Composto por cinco classes distribuídas em dois pacotes, o primeiro projeto recebe o nome da própria ferramenta. Como pode ser observado na Figura 5.3, esta parte do projeto possui uma pequena quantidade de classes com a principal função de exibir as *interfaces* gráficas. Sua ligação com o segundo projeto, nomeado como POS (*Point of Sale*), ocorre através da classe "*GlobalVars*" que fornece os dados previamente informados para geração de eventos. A classe "*WGSysEFT*" apenas possui o método *main*.

O *framework* é de fato utilizado apenas no segundo projeto, pois é neste projeto que reside a classe "*MyAgent*", no qual o método *generatingLoad()* pertencente ao *framework* foi sobrescrito. Esta classe é responsável por receber os dados cedidos pela classe "*GlobalVars*", tomar algumas decisões a respeito do evento a ser criado (pagamento com cartão de crédito ou débito, pagamento à vista ou parcelado) e manter os PDVs ativos. A parcela que cabe ao *framework* nesta ferramenta se refere ao acionamento da ferramenta e criação dos PDV, visto que, conforme apresentado na Figura 4.7, o *framework* se encarrega da criação dos PDVs, que acontecerá apenas uma única vez, deixando a cargo do desenvolvedor manter os PDVs ativos. Devido a impossibilidade de se utilizar *threads* o *framework* aciona a quantidade de PDVs informada pela classe "*GlobalVars*" como aplicações individuais, mas ainda sim sob o seu comando. O *framework* também controla o tempo de execução da ferramenta. Ao final da execução da

ferramenta, o mecanismo de nomeação de processos implementado no *framework* (ver maiores detalhes a respeito desse mecanismo no Apêndice B) auxilia na busca dos PDVs criados para a finalização destes. A interação descrita anteriormente entre os projetos pode ser observada na Figura 5.4.

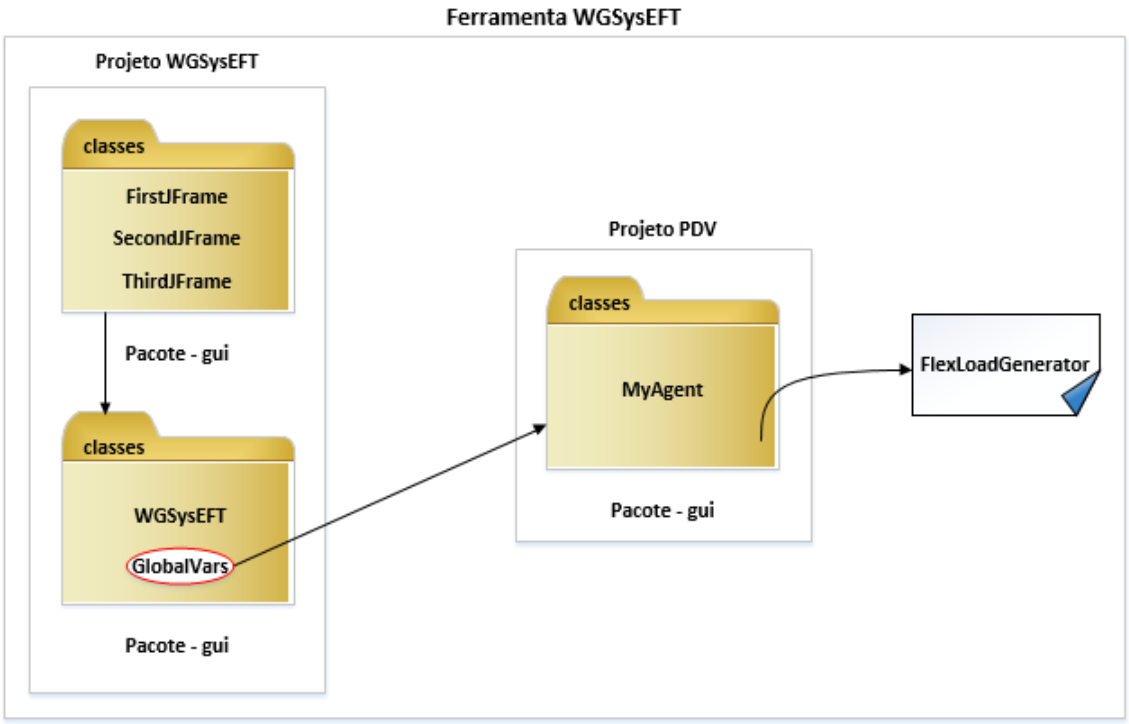


Figura 5.4 Projeto de elaboração da ferramenta WGSysEFT

Observando o projeto como um todo é possível verificar uma redução no tempo e esforço necessário para codificação, visto que o *framework* fornece as distribuições de probabilidade utilizadas para determinar o intervalo de tempo entre a ocorrência de novas transações, criação dos PDVs e o controle sobre a execução da ferramenta.

5.2 EucaBomber

Esta seção descreve brevemente computação em nuvem, apresenta sua importância no contexto atual, e descreve a plataforma de criação e gerenciamento Eucalyptus (Eucalyptus, 2013b). Posteriormente, será apresentado o ferramental proposto para injeção de falhas em ambientes de nuvem, seguido pela forma de implementação desta ferramenta, em detalhes.

5.2.1 Computação em Nuvem

Computação em nuvem foi desenvolvido a partir da combinação de tecnologias como *grid computing*, *cluster computing*, *utility computing* e virtualização (Sousa *et al.*, 2012). A computação em nuvem visa prover meios para que usuários façam uso de recursos de forma dinâmica e escalável em forma de serviços (Borko Furht, 2010), através de *interfaces* padrão e protocolos *Web* (Eucalyptus, 2013a). Estas características permitem que os usuários se concentrem em suas atividades fim.

Uma definição de computação em nuvem é apresentada pelo NIST (*National Institute of Standards and Technology*) (Mell and Grance, 2011), onde computação em nuvem é um modelo que permite acesso ubíquo, conveniente, sob demanda a um *pool* compartilhado de recursos computacionais tais como redes, servidores, armazenamento, aplicações e serviços, onde estes podem ser rapidamente configurados e liberados com baixo esforço de gerenciamento, ou interação com o provedor de serviços.

Computação em nuvem possui um modelo de negócio onde o cliente paga ao provedor da nuvem apenas o que utilizar (Mell and Grance, 2011; Foster *et al.*, 2008). Esta forma de pagamento é semelhante ao de serviços básicos, como eletricidade, gás e água. Em geral, computação em nuvem oferece serviços em três diferentes níveis de abstração: infraestrutura como serviço (IaaS), plataforma como serviço (PaaS) e *software* como serviço (SaaS) (Foster *et al.*, 2008; Neamtiu and Dumitras, 2011).

- **IaaS:** fornece recursos computacionais como armazenamento, processamento e rede em forma serviço (Borko Furht, 2010; Rimal *et al.*, 2009), onde o cliente paga apenas o que utilizar (Foster *et al.*, 2008). Alguns exemplos de IaaS incluem a Amazon EC2 (*Elastic Compute Cloud*) (Amazon, 2013a), e S3 (*Simple Storage Service*) (Amazon, 2013b), além de Google Compute Engine (Google, 2013b).

- **PaaS:** semelhante a IaaS (Borko Furht, 2010), PaaS oferece um ambiente de desenvolvimento onde é possível construir, testar e implantar aplicações (Foster *et al.*, 2008). Neste modelo de negócio o desenvolvedor não controla, por exemplo, a rede ou sistema operacional da nuvem, apenas exerce controle sobre as ferramentas que criar e define a configuração do ambiente para execução de aplicativos (Mell and Grance, 2011). Um exemplo de PaaS é o Google App Engine (Google, 2013a).

- **SaaS:** fornece um conjunto de aplicações que executam na nuvem (Neamtiu and Dumitras, 2011), onde os clientes podem fazer o acesso de forma remota (Foster

et al., 2008). Esta forma de serviço visa substituir *software* que antes eram instalados localmente em máquinas pessoais. Com o SaaS, o cliente paga um valor inferior ao valor de aquisição de *softwares*. Alguns exemplos de SaaS incluem o Salesforce (Salesforce, 2013) e o Google Docs (Google, 2013c).

Além da escolha do modelo de negócio, o usuário pode optar também por um dos diversos tipos de nuvens de acordo as suas necessidades.

- **Nuvem privada:** esse tipo de nuvem é construído pelo próprio provedor para uso exclusivo (Borko Furht, 2010). Quando uma empresa opta por este tipo de nuvem, fica responsável pela gerência tanto de dados como de recursos. Também fica a cargo da empresa manter os serviços em operação, além de modernização de equipamentos ou a manutenção dos mesmos.

- **Nuvem pública:** diferentemente da nuvem privada, as nuvens públicas são criadas e gerenciadas por terceiros, e dados de diferentes clientes coabitam na mesma infraestrutura (Borko Furht, 2010). Os serviços oferecidos por este tipo de nuvem abrangem clientes que não estão dispostos a manter toda uma estrutura computacional para utilizar os serviços providos por uma nuvem. A forma de acesso a nuvem ocorre por meio da Internet. O local onde a infraestrutura da nuvem é mantida não necessita ser de conhecimento dos usuários. A vantagem desse tipo de nuvem reside em oferecer recursos como processamento, armazenamento, entre outros, aos clientes de forma dinâmica e sem a necessidade de se adquirir, atualizar ou realizar qualquer tipo de manutenção em máquinas.

- **Nuvem híbrida:** é a combinação dos modelos de nuvens privadas e públicas. Neste tipo de nuvem tanto a empresa fornecedora do serviço quanto o usuário são responsáveis por manter o serviço em funcionamento.

5.2.2 O *Framework* Eucalyptus e seus Componentes

Eucalyptus (*Elastic Utility Computing Architecture Linking Your Programs To Useful Systems*) é um *software* que implementa estilos escaláveis de IaaS (*Infrastructure as a service*) para nuvens privadas e híbridas (Eucalyptus, 2009), oferecendo *interface* compatível com os serviços EC2 e S3, providos pela Amazon (Eucalyptus, 2013a; Amazon, 2013a). Esta compatibilidade permite executar aplicações tanto na Amazon, como no Eucalyptus sem modificações.

Em geral, a plataforma Eucalyptus se utiliza de virtualização (*hypervisor*) do sistema computacional subjacente para permitir alocação de recursos de forma flexível, dissociado de *hardware* específico. A arquitetura do Eucalyptus é composta de cinco componentes de alto nível, cada um com sua própria *interface* web, são eles: *Cloud Controller* (CLC), *Node Controller* (NC), *Cluster Controller* (CC), *Storage Controller* (SC), e *Walrus*. A Figura 5.5 mostra um exemplo de uma infraestrutura de computação em nuvem, baseada no Eucalyptus, considerando-se dois *clusters*, A e B. Cada *cluster* possui um *Cluster Controller*, um *Storage Controller* e vários *Node Controller*. Os componentes, em cada *cluster*, se comunicam com o *Cloud Controller* e *Walrus* para atender as solicitações do usuário. Uma breve descrição desses componentes pode ser vista a seguir:

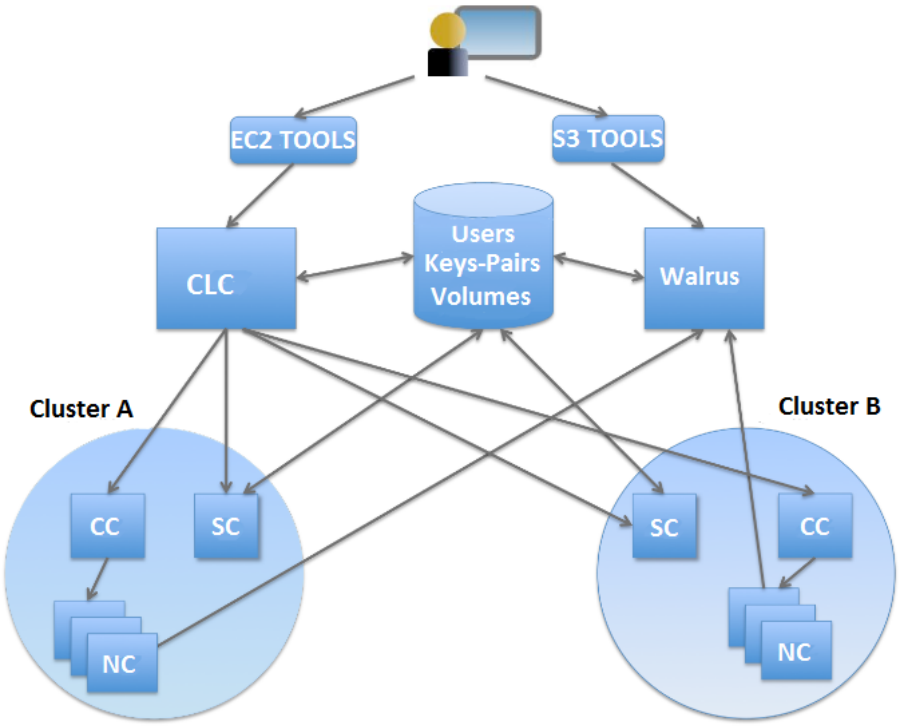


Figura 5.5 Exemplo de ambiente em nuvem gerenciado pelo Eucalyptus

Cloud Controller (CLC): é o ponto de acesso para a infraestrutura da nuvem. Este componente utiliza *interface* web para receber solicitações e interagir com o restante dos componentes (Dantas *et al.*, 2012). O CLC é responsável por monitorar a disponibilidade dos recursos em diversos componentes da infraestrutura, arbitragem de recursos e monitoramento de instâncias em execução (D *et al.*, 2010).

Node Controller (NC): é executado em cada nó (máquina física), gerencia o ciclo de

vida de máquinas virtuais (VMs) em operação no nó. Este componente realiza consultas para descobrir se recursos físicos estão disponíveis, por exemplo, número de núcleos de CPU (*Central Processing Unit*), tamanho da memória principal, e também para sondar o estado de instâncias presente nesse nó.

Cluster Controller (CC): faz o gerenciamento de um ou mais NC. Este componente reúne informações sobre um conjunto de máquinas virtuais e horários de execução de VM no NC específico. O CC tem três funções básicas (D *et al.*, 2010): solicitação de requisição para execução de instância de VMs; controlar a sobreposição de rede virtual composta por um conjunto de VMs, coletando informações sobre um conjunto de nós; e reportar o seu status para o CLC.

Storage Controller (SC): fornece armazenamento permanente para ser usado por instâncias de VMs. Este componente implementa acesso a bloco de armazenamento em rede, semelhante ao proporcionado pela *Amazon Elastic Block Storage* (EBS). As principais funções do SC são: criação persistente de EBS; permitir criação de snapshots de volumes; e fornecer armazenamento de bloco aos protocolos AoE (*ATA over Ethernet*) ou ISCSI (*Internet Small Computer System Interface*) para instâncias.

Walrus: é um serviço de armazenamento baseado em arquivos compatíveis com o *Simple Storage Service* (S3) da Amazon (Eucalyptus, 2009). Pode ser usado por imagens de VMs além de servir como um repositório para arquivos de sistema de VMs, *ramdisk* e imagens do *kernel* Linux, usados para instanciar VMs nos nós físicos da nuvem.

5.2.3 EucaBomber - Visão Geral

O EucaBomber é um gerador de eventos de falhas, com opção de reparo, que emula a ausência de operações no sistema de nuvem gerenciado pela plataforma Eucalyptus. Esta ferramenta foi concebida visando oferecer suporte a estudos de confiabilidade e disponibilidade.

As falhas causadas pela aplicação são transientes, isto é, ela simula um possível estado de interrupção da execução do sistema, de modo que possa ser reparado. As perturbações causadas através da inserção de falhas no sistema de nuvem, visam afetar a execução dos componentes de alto-nível do Eucalyptus, além da infraestrutura física que abriga a nuvem. A ação de reparo objetiva recuperar o sistema de eventuais falhas causadas pelo injetor. O reparo só é executado após a inserção de falhas, o que implica que a ferramenta não faz o reparo de uma falha que não foi injetada por ela mesma.

O disparo de falhas e o reparo destas, quando solicitado, ocorre após um período de tempo determinado de forma aleatória. Maiores detalhes sobre inserção de falhas e

reparos são apresentados a seguir.

O EucaBomber executa eventos nos seguintes modos de operação:

- **Falhas de *hardware*:** este modo operacional emula falhas de *hardware* através do desligamento de dispositivos (por exemplo, servidor, *desktop* etc.);
- **Falhas e reparos de *hardware*:** este modo operacional inclui tanto falhas quanto reparos relacionados ao *hardware*. Após a falha o EucaBomber reinicia novamente a máquina depois de aguardar um tempo de espera. Este modo operacional requer que a máquina alvo ofereça suporte ao padrão WOL (*Wake on Lan*) (Popa and Slavici, 2009). Esta característica é essencial devido a ação de reparo ser realizada por meio do envio de um "pacote mágico" através da rede para iniciar a máquina.
- **Falhas de *software*:** falhas em componentes de alto nível do Eucalyptus, previamente mencionados, estão incluídos neste modo operacional. A ferramenta atua diretamente, suspendendo a execução de um processo do Eucalyptus selecionado pelo usuário. Vale ressaltar que na mesma máquina pode-se injetar mais de um tipo de falha de *software*.
- **Falhas e reparos de *software*:** a diferença entre este modo operacional e o anterior é que este não apenas atua causando falhas, mas também realizando reparo. O reparo é realizado através da restauração dos processos do Eucalyptus que haviam sido anteriormente finalizados pelo EucaBomber.

A ferramenta possibilita a combinação de mais de um modo de operação anteriormente descrito em um experimento. Essa combinação caracteriza os possíveis cenários.

O EucaBomber atualmente se restringe a ambientes que utilizem o sistema operacional Ubuntu Linux. Entretanto, a ferramenta pode ser adaptada para permitir eventos de falhas mais elaborados. Falhas relacionadas com a ação de suspensão da operação do *hardware* também podem ser modificadas. Neste caso, o desenvolvedor pode adicionar outras falhas de *hardware* através de *software*, tais como perda de dados armazenados em *cache*. O manual do usuário contendo maiores informações sobre como utilizar o EucaBomber pode ser visto no Apêndice D.

O EucaBomber possui um núcleo de funções que suporta os meios básicos que são necessários para a construção do injetor. O núcleo que integra a ferramenta é descrito em maiores detalhes a seguir.

5.2.4 *Kernel*

O *Kernel* presente na estrutura do EucaBomber foi implementado a partir do FlexLoadGenerator. Tecnicamente, o *kernel* oferece apenas o meio de comunicação com o sistema alvo e o gerenciamento da ocorrência de eventos.

O *kernel* consiste em duas partes fundamentais: módulo de comunicação e módulo de geração de números aleatórios.

O módulo de conexão é responsável por prover a comunicação entre o injetor de falhas e a máquina alvo. A conexão é estabelecida usando o protocolo SSH2 que permite que comandos sejam enviados diretamente ao *shell* do sistema operacional Ubuntu Linux presente nos servidores que compõem a nuvem. O módulo de geração de números aleatórios é responsável por gerar números pseudo-aleatórios que seguem alguma dentre as distribuições de probabilidade disponíveis. A ferramenta utiliza estes valores como intervalo de tempo (*time-out*) para a ocorrência dos eventos. As distribuição de probabilidade já foram anteriormente detalhas no capítulo 4.

5.2.5 Desenvolvimento do EucaBomber

O processo de desenvolvimento do Eucabomber foi marcado por algumas escolhas de projeto. A primeira escolha foi referente ao protocolo a ser utilizado para estabelecer a comunicação entre a ferramenta e o sistema em teste. O protocolo SSH2 foi o escolhido devido a possibilidade de enviar comandos de causariam falhas de *hardware* e *software*, além dos reparos de *software*. Posteriormente buscou-se uma solução que pudesse fazer reparos nas máquinas que sofreram falhas de *hardware*. A solução escolhida para realizar esta tarefa foi o padrão para rede Ethernet WOL (Wake on Lan) (Popa and Slavici, 2009). A principal justificativa para tal escolha se deu por considerarmos que a ferramenta atuaria na mesma rede de operação do ambiente em nuvem. E, por fim, decidir como o *kernel* poderia ser implementado.

O *kernel* do Eucabomber foi inteiramente construído a partir do *framework*. A base de desenvolvimento do *kernel* é exatamente igual ao exposto na Figura 4.3 e o comportamento seguido pela ferramenta é retratado na Figura 4.6 (capítulo 4), isto devido a algumas decisões de projeto. A forma de implementação do *kernel* levou em consideração alguns aspectos idealizados para o funcionamento do EucaBomber, tais como: (1) a ferramenta acionaria uma quantidade fixa de processos para geração de eventos; (2) Quando reparos fossem selecionados estes estariam associados a suas respectivas falhas, portanto cada processo poderia conter tanto a falha como o reparo desta; e (3) cada

processo possuiria sua própria distribuição e os parâmetros necessários a esta.

O EucaBomber é composto por nove classes que estão distribuídas em seis pacotes. A Figura 5.6 ilustra o projeto do EucaBomber com seus pacotes e classes.

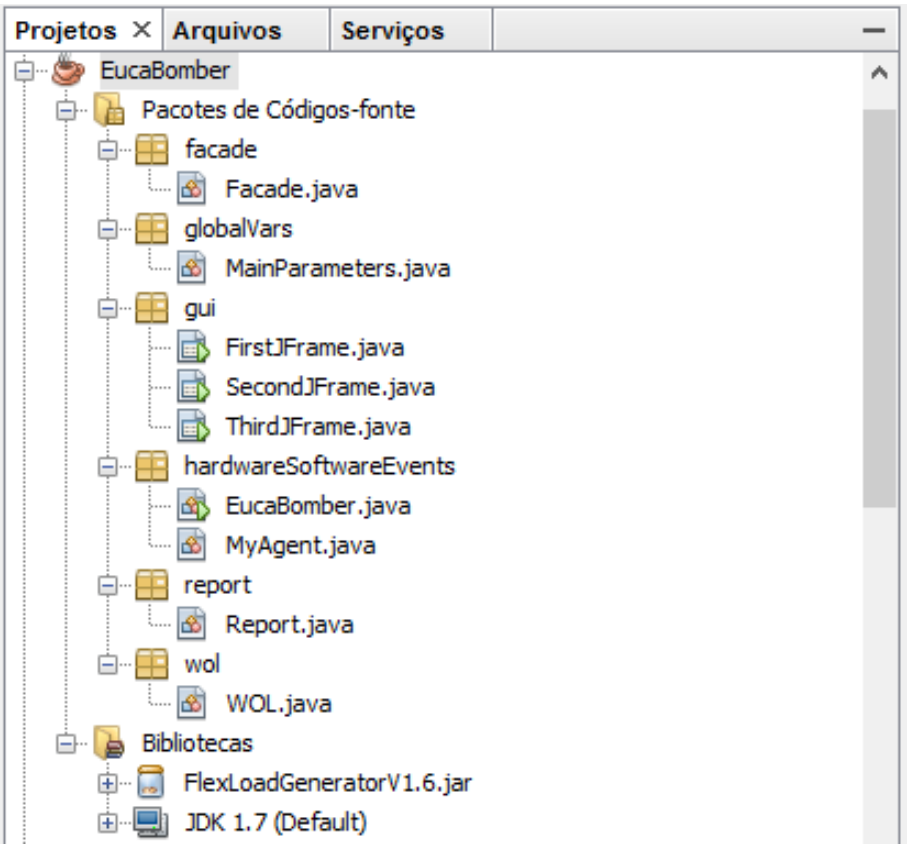


Figura 5.6 *Screenshot* - projeto EucaBomber

A divisão de classes entre pacotes, apresentado na Figura 5.6, foi feita de acordo com o tarefa que cada classe exerce. Classes com funções semelhantes foram agrupadas no mesmo pacote, que é o caso das classes "*FirstJFrame*", "*SecondJFrame*" e "*FirstThird*" que são as interfaces gráficas apresentadas sequencialmente ao usuário a medida que os dados solicitados em cada uma delas vão sendo preenchidos. A classe "*MainParameters*" apenas armazena as escolhas do usuário temporariamente visto que estes são escritos em um arquivo que posteriormente é lido para geração de eventos. Já "*WOL*" é a classe responsável por fazer os reparos de *hardware* enviando "pacotes mágicos" para restabelecer as máquinas afetadas por falhas deste tipo. Os relatórios de execução elaborados pelo EucaBomber são de responsabilidade da classe "*Report*". A classe "*Facade*" trabalha ocultando os detalhes de implementação entre as classes pertencentes aos pacotes "*gui*" e as classes do pacote "*hardwareSoftwareEvents*". O pacote "*hardwa-*

reSoftwareEvents" possui as classes *EucaBomber*, que apenas contém o método *main* e *MyAgent* que é a classe responsável por conectar esta parte da aplicação ao *kernel*. A Figura 5.7 mostra o diagrama de classes com os pacotes (em vermelho) e o relacionamento entre as classes que o constituem o EucaBomber, com exceção do *kernel*. A ligação entre o *kernel* e o restante da aplicação ocorre de forma transparente através da classe "'*MyAgent*".

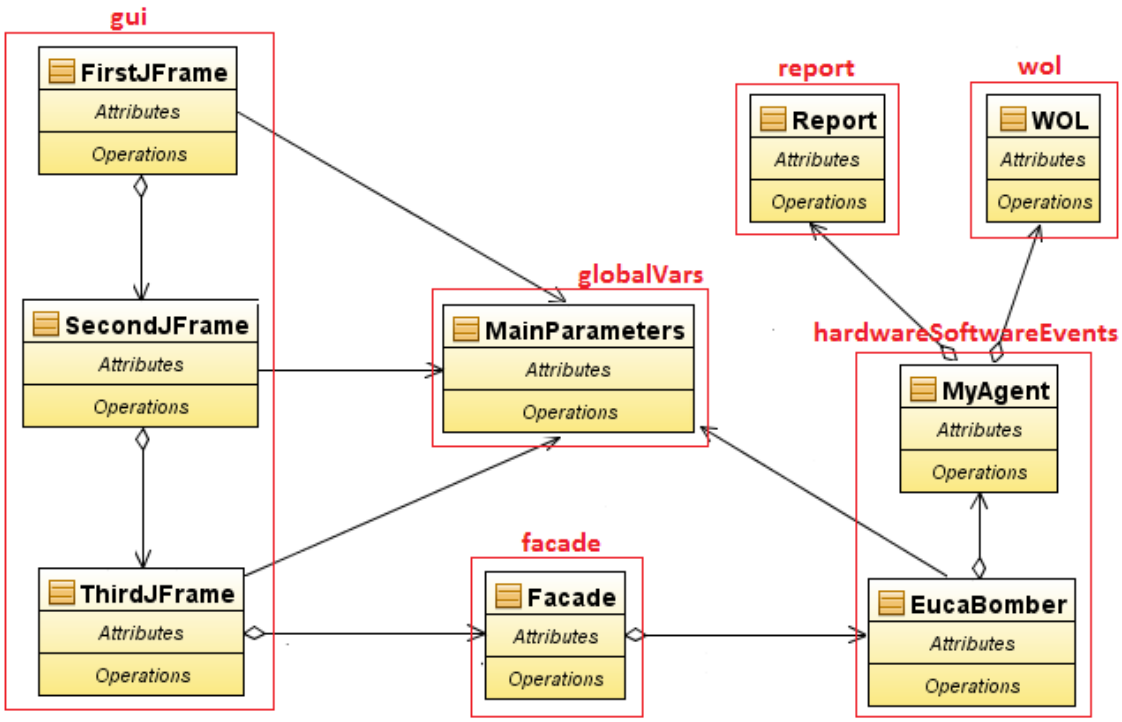


Figura 5.7 Diagrama de classes do projeto EucaBomber

É importante observar que a grande maioria das classes desenvolvidos no projeto do EucaBomber são relacionados apenas a *interface* gráfica e a definição dos eventos a serem criados. Todo o arcabouço necessário para criação de eventos e gerenciamento da ferramenta advém do *framework*, onde este foi utilizado para construir o *kernel* do EucaBomber. As classes anteriormente mencionadas apenas captam as opções selecionadas pelos usuários, escrevem os dados em um arquivo de texto, elaboram o relatório contendo alguns dados dos acontecimentos ao longo da execução da ferramenta e, por fim, mostra ao usuário, através de contagem regressiva, que o tempo da execução da ferramenta está se aproximando do fim ou que de fato chegou ao final. No entanto, o *kernel* é o grande responsável por efetivamente transformar os dados armazenados no arquivo em eventos de falhas e/ou reparos, injetando-os no sistema. Também é de responsabilidade do *kernel* encerrar os processos que geram os eventos ao final da execução

da aplicação.

Neste projeto não houve qualquer esforço no sentido de implementar como a geração de eventos deveria acontecer e como esta seria controlada, reduzindo assim o tempo e os esforços necessários a codificação da ferramenta. Do *framework* foi aproveitado o protocolo para inserção de falhas e reparos (SSH2), as dez distribuições de probabilidade disponibilizadas (ver a relação das distribuições no capítulo 4, subseção 4.4.1), criação de eventos e o gerenciamento da ferramenta ao longo de sua execução.

5.3 Considerações Finais

Este capítulo apresentou as ferramentas desenvolvidas a partir do FlexLoadGenerator. Inicialmente, foi descrito o sistema TEF, abordando tópicos como pagamento por meio eletrônico e o sistema TEF. Em seguida, o WGSysEFT foi detalhado juntamente com sua forma de codificação. Logo após, foi contextualizado o ambiente de computação em nuvem Eucalyptus, e seus componentes. Finalizando o capítulo, foi apresentado o gerador de eventos de falhas e reparos para ambientes de nuvem gerenciados pela plataforma Eucalyptus, intitulado EucaBomber, onde foi descrito sua características e o modo de desenvolvimento da ferramenta.

6

Estudos de Caso

Determinação, coragem e auto confiança são fatores decisivos para o sucesso.

—DALAI LAMA

Neste capítulo são apresentados estudos de caso que objetivam avaliar questões de desempenho e dependabilidade em sistemas, empregando as ferramentas apresentadas no capítulo 5. O intuito principal deste capítulo é demostrar a eficiência das ferramentas geradoras de eventos construídas com o auxílio do FlexLoadGenerator na realização de suas atividades, através de estudos de caso. O primeiro estudo de caso exposto visa avaliar o desempenho de máquinas que abrigam a solução do sistema TEF frente a uma grande quantidade de solicitações e execução de transações. O segundo estudo avalia a disponibilidade de uma infraestrutura gerenciada pela plataforma de computação em nuvem Eucalyptus mediante a inserção de falhas e reparos tanto no *hardware* que compõe a nuvem, como nos principais processos da plataforma Eucalyptus.

6.1 WGSysEFT

Esta seção apresenta um estudo de caso que objetiva avaliar o desempenho de um sistema TEF frente a uma grande quantidade de transações através da utilização do WGSysEFT (*Workload Generator for Electronic Funds Transfer System*) como ferramenta geradora de carga de trabalho sintética. Durante os experimentos, procurou-se observar o comportamento de alguns recursos pertencentes à máquina que abriga o gerenciador de transações TEF. O ambiente de execução de testes foi planejado, visando avaliar o desempenho do sistema com diferentes quantidades de pontos de venda (PDVs) ativos e em máquinas de configurações distintas.

6.1.1 Ambiente de Teste

O ambiente de teste utilizado neste estudo de caso fez uso da infraestrutura virtual fornecida pelo serviço Amazon EC2 (Amazon, 2013a). Para a realização desta pesquisa foram construídas cinco VMs, uma delas do tipo m1.small (Amazon, 2013c) e as demais ao tipo m1.medium (Amazon, 2013c). Em duas das máquinas virtuais, sendo uma do tipo m1.small e a outra do tipo m1.medium, foram instalados o SCOPE (Solução Completa para Pagamento Eletrônico), solução TEF pertencente a Itautec (Itautec, 2013a). As VMs restantes receberam o WGSysEFT. Cada VM corresponde a arquitetura do processador em 32 *bits* e o sistema operacional Microsoft Windows 2008 R1 SP2 Datacenter Edition. A Tabela 6.1 resume a configuração básica das VMs.

Tabela 6.1 Configuração básica das VMs que compõe o ambiente de teste

	Número de máquinas	**Unidade de CPU**	**Núcleo CPU**	**Memória (GB)**	**Armazenamento (GB)**
m1.small	1	1 ECUs[1]	1	1.7	160
m1.medium	4	2 ECUs[1]	1	3.7	410

Um ponto a ser destacado na instalação do sistema TEF é a configuração da forma de operação do SCOPE: modo operação e modo demonstração. O modo de operação é utilizado em estabelecimentos comerciais, enquanto o modo de demonstração é utilizado apenas em ambientes para teste do sistema. Neste trabalho, o sistema TEF foi configurado em modo de demonstração. A arquitetura do ambiente de teste é apresentada na Figura 6.1.

O sistema presente na Figura 6.1 centraliza em uma única máquina o gerenciador de transações TEF (1), o servidor de banco de dados (2) responsável por armazenar as transações e o emulador de empresas autorizadoras (3).

A coleta de dados para medição do desempenho foi realizada através do Windows Performance Monitor (Perfmon) (Perfmon, 2013). O intervalo entre coletas foi definido como 5 segundos. As métricas coletadas, juntamente com sua descrição, estão dispostas na Tabela 6.2.

6.1.2 Descrição dos Cenários

O estudo de caso foi dividido em dois cenários: geração de carga de trabalho sintética baseada na distribuição de probabilidade exponencial, e baseada na distribuição

[1]Onde, ECU significa *EC2 Compute Units*. Uma ECU fornece uma CPU com capacidade equivalente a 1.0-1.2 GHz 2007 Opteron ou Xeon 2007

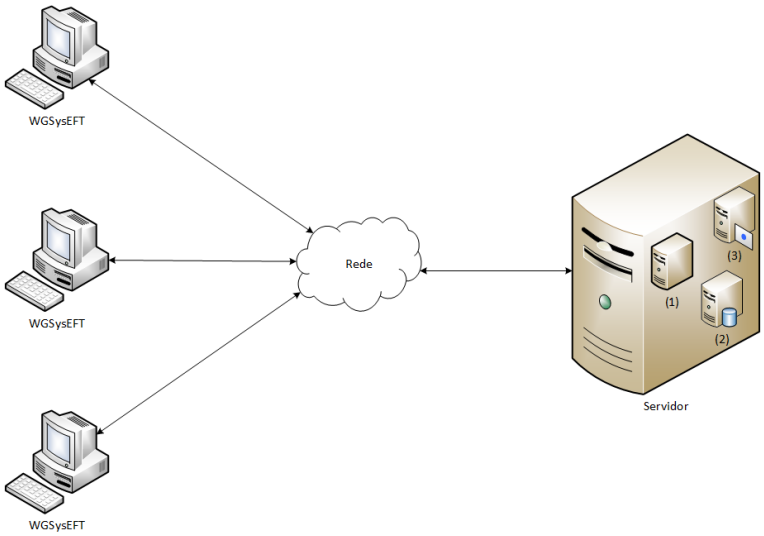


Figura 6.1 Ambiente de teste utilizado para experimentos com o WGSysEFT

Erlang, denominados daqui por diante, cenários 1 e 2, respectivamente. A diferenciação dos cenários visa compreender a influência dos tempos entre requisições a partir de diferentes distribuições de probabilidade. Ambos os cenários têm por finalidade observar a variação no consumo dos recursos computacionais mencionados na Tabela 6.2 e como o aumento do número de PDVs em operação impacta no aumento do consumo. Para que este objetivo fosse alcançado optou-se por compor ambos os cenários com os tipos de VMs **m1.small** e **m1.medium** e a quantidade de PDVs utilizados em 100, 150 e 200.

O tempo de duração total dos experimentos correspondeu a 12 horas, onde cada cenário foi executado durante 6 horas. Das 6 horas destinadas a cada cenário, 3 delas foram dedicadas a execução dos experimentos em VM do tipo m1.small e mais 3 horas a VM do tipo m1.medium. E, por fim, as quantidades de PDVs foram testadas durante 1 hora cada.

Alguns valores informados a ferramenta foram mantidos inalterados em ambos os cenários. São eles:

- **Tipo de pagamento:** crédito e débito;
- **Forma de pagamento:** à vista e parcelado, sendo este último apenas para crédito.
- **Percentuais de transações:** 34% para transações de crédito à vista e 33% para as demais, totalizando assim 100%;

Um outro fator que diferencia os cenários é a distribuição de probabilidade utilizada e os seus respectivos parâmetros para geração de valores que são utilizados como intervalo

Tabela 6.2 Métricas selecionadas juntamente com sua descrição

Métrica	Descrição
Utilização do processador (%)	Principal indicador de atividade do processador. Esta métrica contabiliza o percentual médio de utilização em um determinado intervalo de tempo de ocupação do processador.
Utilização de processador – processo servidor TEF (%)	Percentual médio de utilização do processador decorrido em um determinado intervalo de tempo em que todas as *threads* referentes ao processo gerenciador TEF fizeram uso do processador para executar instruções.
Utilização de processador – processo autorizador TEF (%)	Percentual médio de utilização do processador decorrido em um determinado intervalo de tempo em que todas as *threads* referentes ao processo autorizador TEF fizeram uso do processador para executar instruções.
Utilização de disco (%)	Percentual médio de utilização em um determinado intervalo de tempo de ocupação na qual a unidade de disco estava ocupada atendendo solicitações de leitura e gravação.
***Bytes* de gravação em disco (s)**	Taxa na qual *bytes* são transferidos para gravação em disco.
Memória disponível (MB)	Quantidade de memória física, em MB, disponível para alocação.

de tempo entre o final de uma transação e o início da próxima. O cenário 1, baseado na distribuição exponencial, recebeu como parâmetro o tempo médio igual à 1000ms (cuja taxa configurada para a distribuição exponencial foi igual a 0,001). Já o cenário 2 que é baseado na distribuição Erlang recebeu como parâmetros o tempo médio de 1000ms (taxa 0,001) e forma (k) no valor k=5. A opção por informar estes parâmetros às distribuições escolhidas para cada cenário se deve a testes anteriormente realizados visto que a pretensão era atingir no mínimo 45% de processamento em ambos os tipos de máquinas virtuais utilizadas nos testes.

Um ponto a ser destacado está relacionado aos percentuais estabelecidos para os parâmetros selecionados referente às formas de pagamento. Tais percentuais foram assim definidos devido à dificuldade de se obter dados dos estabelecimentos comerciais com os percentuais de tipo de pagamento que ocorreram em um dado intervalo de tempo, seja ele dia, semana, mês ou qualquer outro período. Por esse motivo optou-se por distribuir as transações em proporções aproximadamente iguais para cada um dos 3 tipos: débito à vista, crédito à vista, e crédito parcelado.

Outra observação a ser feita consiste em não se ter escolhido a forma de pagamento "*forward purchase*", que permitiria pagamentos na categoria débito parcelado. Esta opção não foi escolhida para atuar em nenhum dos cenários devido ao gerenciador de

transações TEF utilizado nos testes não oferecer suporte a este tipo de operação.

Os cenários anteriormente mencionados, assim como os resultados obtidos em cada um são detalhados na subseção a seguir.

6.1.3 Estudo de Caso - Geração de Carga de Trabalho para Sistemas TEF com Base em Distribuições de Probabilidade

Esta seção apresenta os resultados do estudo de caso para ambos os cenários descritos da subseção 6.1.2. Para cada métrica selecionada foram coletadas e analisadas 720 amostras, visto que as quantidades de PDVs anteriormente estabelecidas para realização dos experimentos executaram por 1 hora cada e o intervalo entre coletas foi estipulado em 5 segundos. As Tabelas 6.3 e 6.4 apresentam à média, coeficiente de variação, desvio padrão e o intervalo de confiança (em 95%) em relação a média de utilização da métrica "Utilização do Processador (%)" para os cenários 1 e 2 respectivamente.

Tabela 6.3 Utilização do processador (%) no cenário 1

Quant. PDV	Métrica	m1.small	m1.medium
100	**Média (%)**	70,58	64,05
	Coeficiente de variação	14,11	19,08
	Desvio Padrão	9,92	12,22
	Intervalo de confiança	[69,61; 71,06]	[63,16; 64,95]
150	**Média (%)**	85,49	69,30
	Coeficiente de variação	13,77	13,18
	Desvio Padrão	11,77	9,13
	Intervalo de confiança	[84,63; 86,35]	[68,63; 69,97]
200	**Média (%)**	88,18	73,11
	Coeficiente de variação	14,30	14,72
	Desvio Padrão	12,61	10,76
	Intervalo de confiança	[87,26; 89,11]	[72,32; 73,89]

Em todos os cenários analisados, e suas respectivas variantes, o aumento da quantidade de PDVs acarretou no aumento do uso do processador na máquina que abriga a solução TEF, conforme esperado. No entanto, o processamento diminui visivelmente entre os cenários 1 e 2 devido ao cenário 1 ser baseado na distribuição exponencial enquanto o cenário 2 utiliza a distribuição Erlang. Embora ambas as distribuições utilizem o mesma tempo médio de 1000ms (taxa 0,001), um fator que contribuiu fortemente para essa diferença é o parâmetro forma (*shape*) presente na distribuição Erlang, que recebeu $k=5$. O espalhamento provocado pelo parâmetro de forma (*shape*) desloca mais amostras para valores menores do que o parâmetro taxa (*rate*).

Tabela 6.4 Utilização do processador (%) no cenário 2

Quant. PDV	Métrica	m1.small	m1.medium
100	**Média (%)**	62,69	48,42
	Coeficiente de variação	17,65	15,97
	Desvio Padrão	11,07	7,73
	Intervalo de confiança	[61,88; 63,49]	[47,85; 48,98]
150	**Média (%)**	76,54	54,47
	Coeficiente de variação	20,81	16,96
	Desvio Padrão	15,93	9,24
	Intervalo de confiança	[75,38; 77,70]	[53,80; 55,15]
200	**Média (%)**	81,46	61,04
	Coeficiente de variação	14,91	15,71
	Desvio Padrão	12,15	9,59
	Intervalo de confiança	[80,57; 82,34]	[60,34; 61,74]

É compreensível a discrepância entre as médias referentes aos tipos de VMs, uma vez que o tipo m1.medium possui uma ECU a mais que o tipo m1.small, conforme configuração apresentada na Tabela 6.1. A diferença apresentada em comparações feitas com as médias obtidas entre os dois tipos de VMs, no mesmo cenário e com a mesma quantidade de PDVs resulta em 9%, 18,9%, 17%, 22,7%, 28,8% e 25%, respectivamente na ordem em que aparecem nas Tabelas 6.3 e 6.4. Dentre os percentuais apresentados, é possível observar que o cenário 2 (22,7%, 28,8% e 25%) atinge diferenças de percentuais superiores quando comparado aos apresentados pelo cenário 1. Estes valores mostram, mais uma vez, que o intervalo entre requisições regido pela distribuição Erlang, com os parâmetros informados, resulta em um comportamento consideravelmente distinto do comportamento observado com a distribuição exponencial. Um outra observação pode ser feita quanto a diferença de percentual apresentada acima, a medida que o número de PDVs foi acrescido de 150 para 200 houve uma ligueira queda entre os percentuais de 18,9% (150 PDVs) para 17% (200 PDVs) no cenário 1 e de 28,8% (150 PDVs) para 25% (200 PDVs) no cenário 2. Esta ligueira queda no percentual de 200 PDVs, em comparação com 150 PDVs, em ambos os cenários, é resultado da quase saturação das VMs, que continham o sistema TEF, em responder a grande quantidade de solicitações de transações feitas pelo WGSysEFT.

As Tabelas 6.3 e 6.4 ainda apresentam um resumo estatístico considerando a métrica de utilização do processador. Assim, podemos extrair informações relevantes com relação a homogeneidade dos dados medidos. O coeficiente de variação é uma medida de dispersão que fornece a variação dos dados obtidos em relação à média. Assim, quanto

menor for o seu valor, mais homogêneos serão os dados (EBAPE / Tenório, 2004). O coeficiente de variação é considerado pequeno (apontando um conjunto de dados homogêneos) quando for menor ou igual a 20%. Podemos concluir a partir das Tabelas 6.3 e 6.4 por meio do coeficiente de variação que os dados tratam-se de amostras homogêneas pois estão abaixo de 20%, com exceção do cenário 2 com 150 PDVs (Tabela 6.4) que superou os 20% portanto suas amostras são consideradas heterogêneas. Adicionalmente, verifica-se que o valor resultante do cálculo do intervalo de confiança mostra que o erro de estimativa, nos cenários 1 e 2 cenários, é relativamente pequeno.

As Figuras 6.2, 6.3 e 6.4 mostram a utilização do processador em relação ao cenário 1 na VM do tipo m1.small com 100 (Figura 6.2), 150 (Figura 6.3) e 200 PDVs (Figura 6.4).

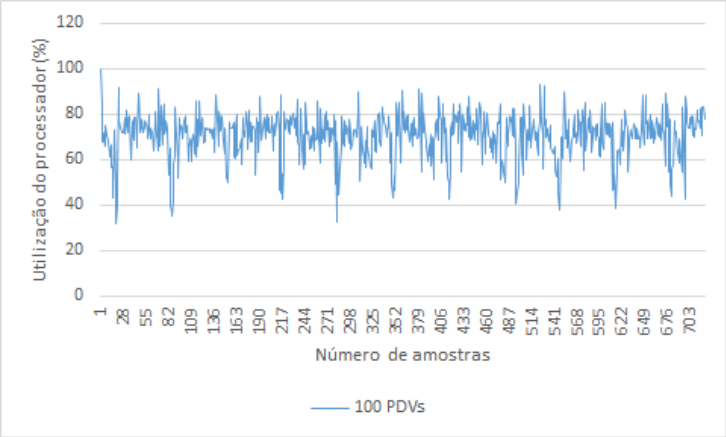


Figura 6.2 Utilização de processador no cenário 1, VM tipo m1.small e 100 PDVs

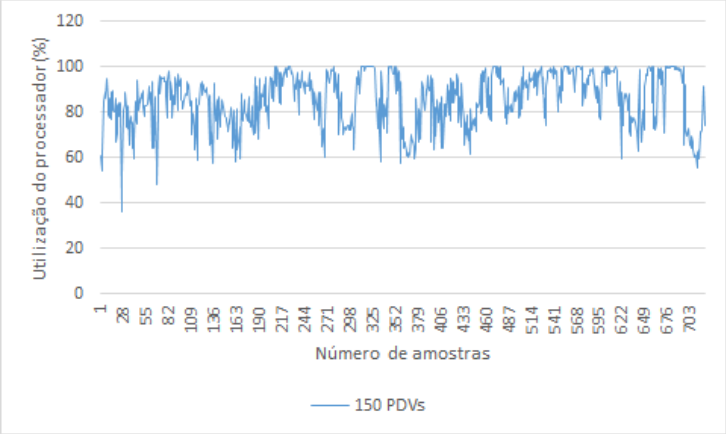


Figura 6.3 Utilização de processador no cenário 1, VM tipo m1.small e 150 PDVs

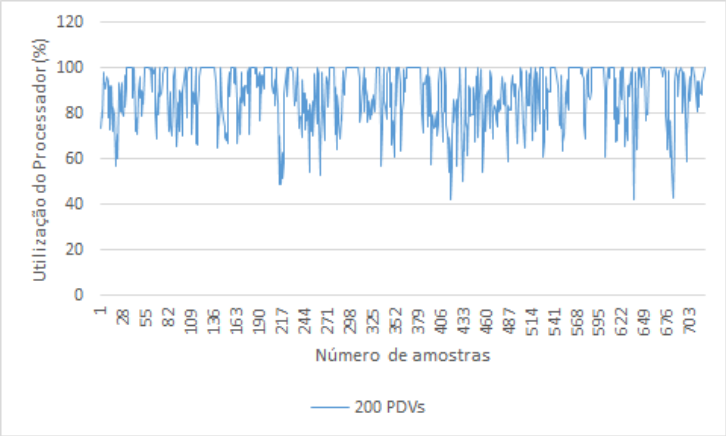


Figura 6.4 Utilização de processador no cenário 1, VM tipo m1.small e 200 PDVs

É preciso observar também que os resultados apresentados nas Tabelas 6.3 e 6.4 não correspondem apenas ao tempo de processamento devido ao gerenciador de transações TEF, mas também ao tempo de processamento do emulador que aprova as solicitações de pagamentos feitas ao sistema. As Tabelas 6.5 e 6.6 apresentam os resultados referentes à utilização do processador, diferenciando o nível de utilização do gerenciador TEF e da aplicação que emula a aprovação de pagamentos.

Como é possível observar, nos cenários 1 e 2 (Tabelas 6.5 e 6.6), o processo responsável pelo gerenciador de transações TEF utilizou mais de processamento do que o processo referente à aplicação que emula a aprovação de transações. Essa evidência pode ser constatada pela diferença entre a utilização do processador devido aos processos, que é superior a 77% no cenário 1 e 79% no cenário 2.

Um outro ponto a ser observado é em relação as médias apresentadas nas Tabelas 6.5 e 6.6 referentes ao processo do Autorizador TEF correspondentes a 150 e 200 PDVs, em ambos os cenários. O aumento do número de PDVs de 150 para 200 faz com que o gerenciador TEF fique mais sobrecarregado para responder todas as trocas de mensagens necessárias para concluir as transações solicitadas com sucesso. Devido a sobrecarga, o gerenciador de transações TEF tende a demorar mais tempo para responder a solicitações, e com isto demora a requisitar a autorização do emulador de transações TEF, visto que este é uma das últimas mensagens a serem trocadas entre o sistema TEF e a ferramenta para finalização de transações. Este processo de troca de mensagens para conclusão de transações com sucesso foi exposto no capítulo 5, com a diferença que ao invés do gerenciador TEF se comunicar com empresas autorizadoras reais este se comunica o emulador de transações.

Tabela 6.5 Utilização do processador pelo servidor TEF e pelo emulador responsável por aprovação de transações no cenário 1

Tipo de VM	Quant. PDVs	Métrica	Servidor TEF	Autorizador TEF
m1.small	100	Média	48,33	7,87
		Coeficiente de variação	15,96	25,01
		Desvio padrão	7,71	1,97
		Intervalo de confiança	[47,77; 48,89]	[7,73; 8,01]
	150	Média	54,84	10,99
		Coeficiente de variação	19,12	30,30
		Desvio padrão	10,49	3,33
		Intervalo de confiança	[54,08; 55,61]	[10,74; 11,23]
	200	Média	61,83	9,87
		Coeficiente de variação	16,62	25,03
		Desvio padrão	10,28	2,47
		Intervalo de confiança	[61,08; 62,59]	[9,69; 10,06]
m1.medium	100	Média	43,51	7,23
		Coeficiente de variação	21,14	22,63
		Desvio padrão	5,28	1,63
		Intervalo de confiança	[43,12; 43,89]	[7,11; 7,35]
	150	Média	50,09	7,68
		Coeficiente de variação	14,11	26,08
		Desvio padrão	7,07	2,00
		Intervalo de confiança	[49,58; 50,61]	[7,53; 7,83]
	200	Média	54,49	5,84
		Coeficiente de variação	15,39	31,19
		Desvio padrão	8,39	1,82
		Intervalo de confiança	[53,88; 55,10]	[5,71; 5,98]

Para essas métricas também foram calculados o desvio padrão e o intervalo de confiança em relação à média de utilização obtida nos processos servidor TEF e o autorizador TEF.

O coeficiente de variação apresentado nas Tabelas 6.5 e 6.6 mostram que as amostras dos cenários 1 e 2 referentes à métrica "'servidor TEF" são mais dispersas no cenário 1 apenas com 100 PDVs na VM do tipo m1.medium e no cenário 2 na VM do tipo m1.small com 150 PDVs. A métrica "autorizador TEF" foi superior a 20% em ambos os cenários mostrando a dispersão das amostras. Portanto, para esta métrica todos os grupos de amostras dos cenários 1 e 2 são heterogêneas.

Tabela 6.6 Utilização do processador pelo servidor TEF e pelo emulador responsável por aprovação de transações no cenário 2

Tipo de VM	Quant. PDVs	Métrica	Servidor TEF	Autorizador TEF
m1.small	100	Média	43,30	6,78
		Coeficiente de variação	18,90	29,14
		Desvio padrão	8,18	1,97
		Intervalo de confiança	[42,70; 43,89]	[6,63; 6,92]
	150	Média	54,91	7,70
		Coeficiente de variação	21,29	39
		Desvio padrão	11,69	3,00
		Intervalo de confiança	[54,06; 55,77]	[7,48;7,92]
	200	Média	58,46	6,24
		Coeficiente de variação	15,47	33,89
		Desvio padrão	9,04	2,11
		Intervalo de confiança	[57,80; 59,12]	[6,08; 6,39]
m1.medium	100	Média	34,46	6,17
		Coeficiente de variação	17,09	26,97
		Desvio padrão	5,89	1,66
		Intervalo de confiança	[34,03; 34,89]	[6,05; 6,29]
	150	Média	39,89	5,93
		Coeficiente de variação	19,38	31,76
		Desvio padrão	7,73	1,88
		Intervalo de confiança	[39,33; 40,46]	[5,79; 6,07]
	200	Média	44,34	7,29
		Coeficiente de variação	16,65	25,68
		Desvio padrão	7,38	1,87
		Intervalo de confiança	[43,80; 44,88]	[7,15; 7,43]

As Figuras 6.5, 6.6 e 6.7 apresentam alguns gráficos que demonstram as métricas ao longo de 1 hora de experimento, para cada PDV que compõem o cenário 1 com VM do tipo m1.small com 100 (Figura 6.5), 150 (Figura 6.6) e 200 PDVs (Figura 6.7), em relação a utilização do processador para as métricas “Processo Servidor TEF” e “Processo Autorizador TEF”.

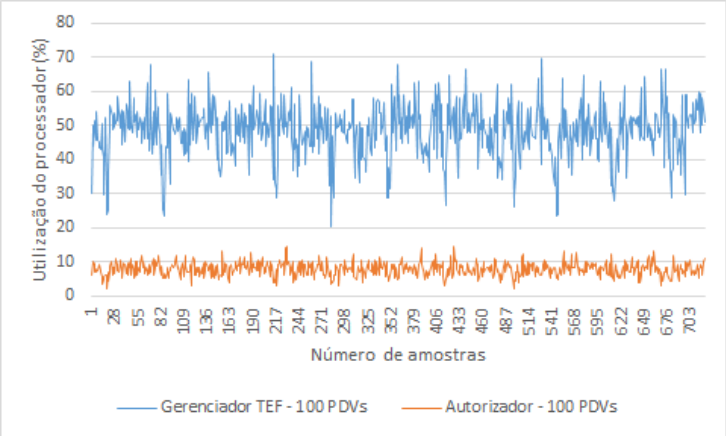


Figura 6.5 Processos servidor e autorizador TEF. Cenário 1, VM tipo m1.small e 100 PDVs

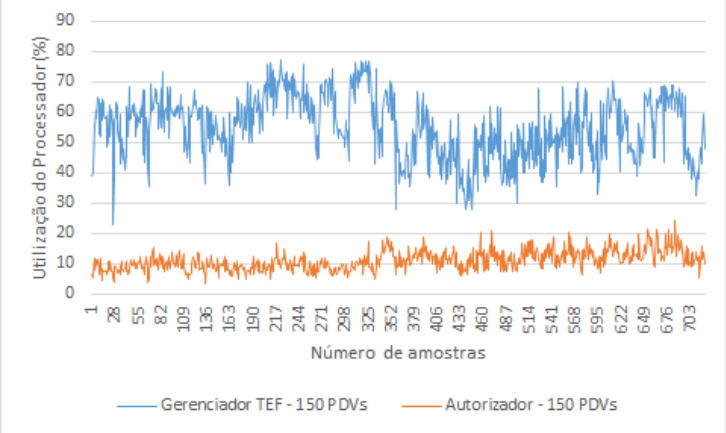


Figura 6.6 Processos servidor e autorizador TEF. Cenário 1, VM tipo m1.small e 150 PDVs

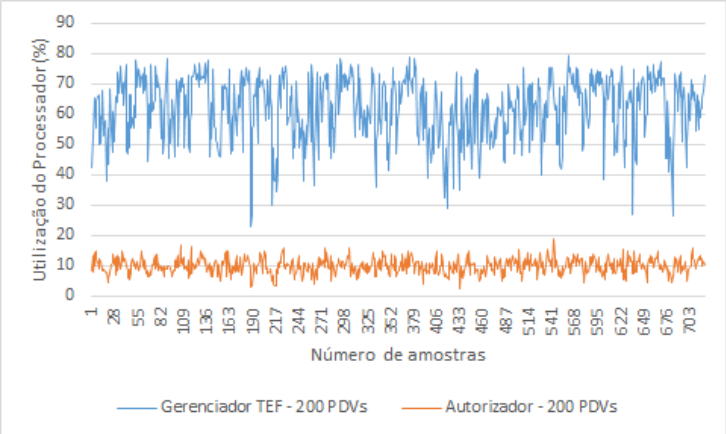


Figura 6.7 Processos servidor e autorizador TEF. Cenário 1, VM tipo m1.small e 200 PDVs

Os resultados obtidos para as métricas “Disco %” e “*Bytes* escritos em disco/s” estão presentes nas Tabelas 6.7 e 6.8.

Tabela 6.7 Métricas “Disco %” e “*Bytes* escritos em disco/s” no cenário 1

Tipo de VM	Quant. PDVs	Métrica	Disco (%)	Bytes escritos em disco/s
m1.small	100	Média	51,56	2205611,73
		Coeficiente de variação	21,46	12,04
		Desvio padrão	11,06	265692,22
		Intervalo de confiança	[50,76; 52,37]	[2186204,63; 2225018,83]
	150	Média	72,27	2023835,62
		Coeficiente de variação	24,86	15,27
		Desvio padrão	17,96	309211,75
		Intervalo de confiança	[70,96; 73,58]	[2001249,69; 2046421,54]
	200	Média	87,76	2181237
		Coeficiente de variação	29,33	12,57
		Desvio padrão	25,74	274299,60
		Intervalo de confiança	[85,88; 89,64]	[2161201,64; 2201273,27]
m1.medium	100	Média	49,27	2932953
		Coeficiente de variação	14,69	11,90
		Desvio padrão	7,23	349139,55
		Intervalo de confiança	[48,74; 49,80]	[2907450,59; 2958455,37]
	150	Média	56,85	3323603,67
		Coeficiente de variação	13,70	13,11
		Desvio padrão	7,79	435810,27
		Intervalo de confiança	[56,28; 57,42]	[3291770,54; 3355436,8]
	200	Média	64,47	3788734,66
		Coeficiente de variação	20,05	15,21
		Desvio padrão	12,93	576571,27
		Intervalo de confiança	[63,53; 65,42]	[3746619,85; 3830849,47]

Conforme esperado, a média do disco rígido para solicitações de leitura e gravação teve seu menor nível no cenário 1 (100 PDVs, tipo m1.medium) como cerca de 49%. Para este mesmo cenário a taxa de *bytes* transferidos para gravação por segundo alcançou a média de 2932953 B/s (2864,21 KB/s). O cenário com maior percentual médio de solicitações de leitura e escrita foi o cenário 2 (200 PDVs, tipo m1.small) com aproximadamente 89% e taxa de transferência de *bytes* com média de 2426173 B/s (2369,30 KB/s). Em ambos os cenários, o uso do disco é menor na VM m1.medium do que na VM m1.small. Estes dados mostram que a capacidade de processamento apresentada pela VM influi na taxa de solicitações de leitura e escrita do disco.

Tabela 6.8 Métricas "Disco %" e "*Bytes* escritos em disco/s" no cenário 2

Tipo de VM	Quant. PDVs	Métrica	Disco (%)	Bytes escritos em disco/s
m1.small	100	Média	50,79	2035974,65
		Coeficiente de variação	25,53	16,74
		Desvio padrão	12,97	340930,12
		Intervalo de confiança	[49,84; 51,74]	[2011071,90; 2060877,39]
	150	Média	85,37	2131347,63
		Coeficiente de variação	24,34	15,08
		Desvio padrão	21,67	255841,77
		Intervalo de confiança	[83,78; 86,95]	[2407485,16; 2444860,34]
	200	Média	89,42	2426173
		Coeficiente de variação	25,39	10,54
		Desvio padrão	21,76	321437,72
		Intervalo de confiança	[87,83; 91,01]	[2107868,68; 2154826,58]
m1.medium	100	Média	55,09	2597296,94
		Coeficiente de variação	16,93	16,29
		Desvio padrão	9,33	423130,82
		Intervalo de confiança	[54,41; 55,77]	[2566389,96; 2628203,91]
	150	Média	61,69	2905836,02
		Coeficiente de variação	20,76	16,56
		Desvio padrão	12,80	481295,41
		Intervalo de confiança	[60,76; 62,63]	[2870680,50; 2940991,54]
	200	Média	71,06	3267224,19
		Coeficiente de variação	21,09	16,39
		Desvio padrão	14,98	535681,50
		Intervalo de confiança	[69,96; 72,15]	[3228096,12; 3306352,27]

O aumento da taxa de leitura e escrita é devido ao incremento do número de PDVs (100 para 200) mesmo considerando-se o mesmo cenário e o mesmo tipo de VM, é de aproximadamente 41,24% para o cenário 1 (m1.small), 23,57% para o cenário 1 (m1.medium), 40,49% para o cenário 2 (m1.small) e 22,46% para o cenário 2 (m1.medium). O comportamento observado nos resultados obtidos na métrica "disco (%)" mostram que o aumento do número de PDVs (de 100 a 200) acarreta um maior tráfego de leitura e escrita de disco.

Da mesma forma que as métricas referentes ao processador, ver Tabela 6.2, também foram calculados o desvio padrão e o intervalo de confiança para as métricas de disco. O coeficiente de variação para métrica "disco(%)" mostra que os únicos dados homogêneos (abaixo de 20%) são apresentados na VM do tipo m1.medium com 100 e 150 PDVs respectivamente. Enquanto os demais são dados heterogêneos devido ao coeficiente de

variação ser superior a 20%. Por outro lado, todas as amostras pertencentes à métrica "*Bytes* escritos em disco/s" são homogêneas ficando abaixo de 20%.

As Figuras 6.8, 6.9 e 6.10 apresenta o comportamento relativo à utilização de disco no cenário 1 com a VM do tipo m1.small com 100 (Figura 6.8), 150 (Figura 6.9) e 200 PDVs (Figura 6.10).

O uso de memória, embora tenha sido uma das métricas selecionadas para estudo, apresentou resultados pouco significativos devido ao uso quase que inalterado de memória nos cenários 1 e 2. A média do uso de memória no cenário 1 (com 100, 150 e 200 PDVs) na VM do tipo m1.small foi de 461 MB. Para este mesmo cenário com o tipo m1.medium o intervalo foi de 2218 MB. O cenário 2 contou com diferenças de 457 MB para o tipo m1.small e 2311MB para o tipo m1.medium.

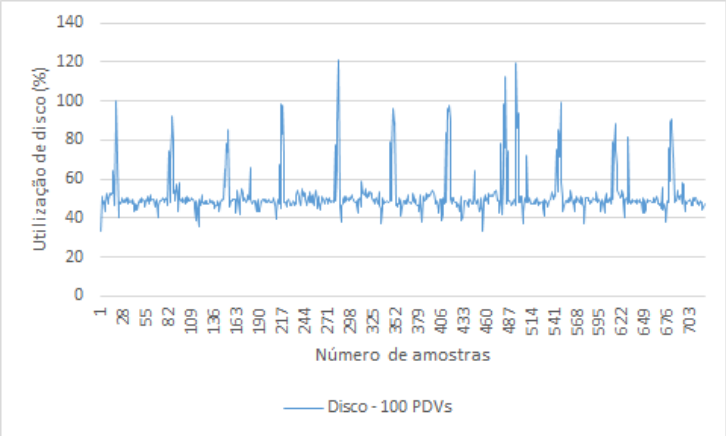


Figura 6.8 Métrica "disco (%)" no cenário 1, VM tipo m1.small e 100 PDVs

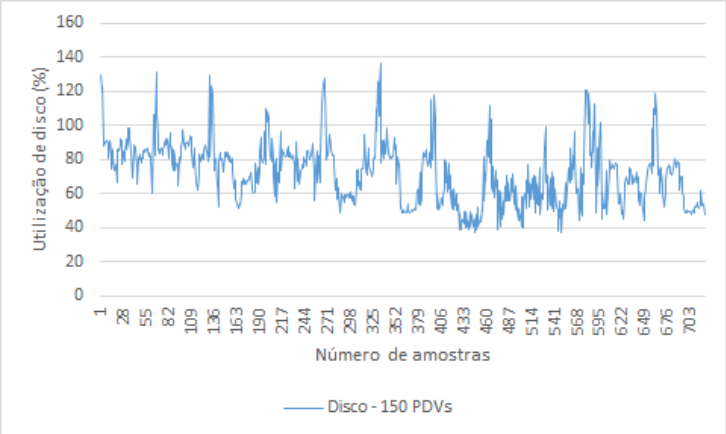


Figura 6.9 Métrica "disco (%)" no cenário 1, VM tipo m1.small e 150 PDVs

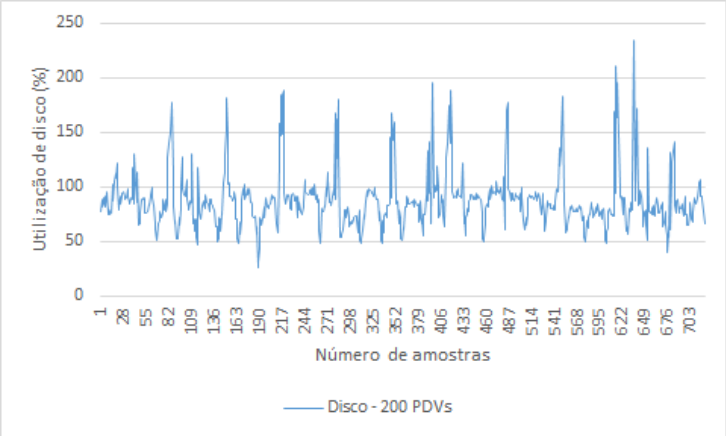


Figura 6.10 Métrica "disco (%)" no cenário 1, VM tipo m1.small e 200 PDVs

6.2 EucaBomber

Esta seção apresenta um estudo de caso envolvendo a utilização do EucaBomber que tem por objetivo a injeção de eventos de falhas e reparos em nuvens gerenciadas pela plataforma Eucalyptus. Este estudo foca em: (1) verificar se os valores dos tempos gerados para ação de falhas e reparos coincidem com as distribuições selecionadas e parâmetros informados e (2) verificar o impacto de distintos MTTFs (*Mean Time to Failure*) e MTTRs (*Mean Time to Repair*) na disponibilidade do sistema em nuvem.

Para realizar os experimentos, foram definidos cenários para observar como a interferência causada por falhas impacta na disponibilidade dos serviços providos por ambientes de nuvem. Neste estudo de caso, a ferramenta também é utilizada para reparar as falhas que injetar. O ambiente de teste no qual experimentos foram realizados e a descrição dos cenários de testes que compõem este estudo de caso são abordados nas subseções a seguir.

6.2.1 Ambiente de Teste

O ambiente de teste onde o EucaBomber foi empregado conta com uma nuvem privada composta por seis máquinas com processador Core 2 Quad de 2.66 GHz e 4GB RAM (*Random Access Memory*). Em cinco máquinas foram instaladas servidores Ubuntu 11.04 (kernel 2.6.38-8 x86-64) e a plataforma Eucalyptus na versão 2.0.2. A sexta máquina foi utilizada como cliente da nuvem e recebeu o sistema operacional Ubuntu para desktop 11.04 (kernel 2.6.38-8 x86-64). A nuvem utilizada como teste é inteiramente baseada na plataforma Eucalyptus e utilizou-se do *hypervisor* KVM.

A Figura 6.11 mostra a arquitetura da nuvem utilizada para testes. A nuvem tem uma estrutura simples, composta por um único *cluster*, com um controlador e quatro nós. Portanto, instalou-se o Cloud Controller (CLC), o Cluster Controller (CC), o Walrus e o Storage Controller (SC) numa mesma máquina física (*host2*). Os Nodes Controllers (NC) foram instalados um em cada nó, pois estes têm que gerenciar os recursos das máquinas físicas (nós) para as respectivas VMs instaladas. Na máquina cliente instalou-se o EucaBomber. É a partir dela que falhas são injetadas no ambiente de teste e os respectivos reparos executados.

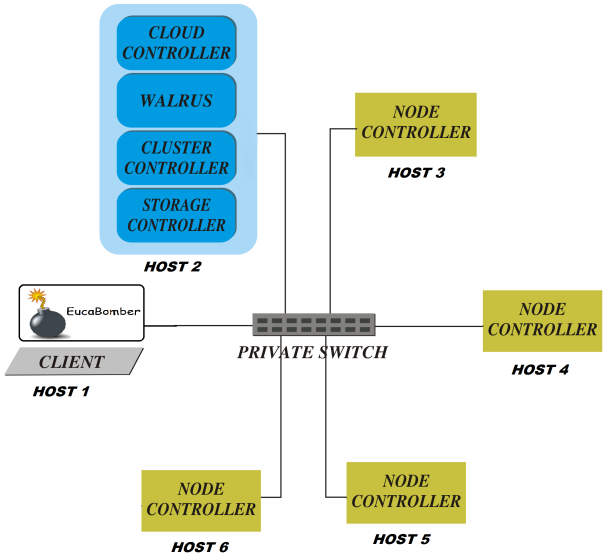


Figura 6.11 Ambiente de teste Eucabomber e seus componentes

A monitoração é feita através de um *script* implementado para verificar a cada 5 segundos se a nuvem está disponível. O *script* assume que a nuvem está disponível se: (1) as cinco máquinas físicas estiverem em operação; (2) os processos CC e CLC estiverem ativos, e; (3) quando os quatro nós estiverem disponíveis para executar VMs. Este *script* também é executado na máquina cliente.

6.2.2 Descrição dos Cenários

O estudo de caso apresentado é composto por quatro cenários onde, os dois primeiros cenários (1a e 1b) são baseados em tempos de falhas e reparos, determinados através da distribuição exponencial. Os tempos dos demais cenários (2a e 2b) adotam a distribuição Erlang.

Independentemente de qual distribuição se tenha adotado, definiram-se alguns parâmetros que foram compartilhados por todos os cenários avaliados. Estes parâmetros são

detalhados a seguir:

- **Tipos de falhas:** *hardware* e *software*;
- **Reparos selecionados:** sim, em ambos os tipos de falhas e em todos os cenários;
- **Quantidade de falhas adicionadas:** onze;
- **Descrição das falhas:** de *hardware*, nas cinco máquinas que compõe a infraestrutura da nuvem; quatro ações de falhas de *software* nas máquinas que possuem o componente Node Controller; uma ação de falha para os componentes Cloud Conroller e Cluster Controller;
- **Quantidade de reparos adicionados:** onze;
- **Descrição dos reparos:** referentes as falhas anteriormente selecionadas;
- **Tempo de duração dos experimentos:** 48 horas (2 dias), para cada cenário, totalizando 192 horas (8 dias); e
- **Total de amostras recolhidas:** 34.560.

Os parâmetros diferenciados fornecidos à ferramenta para cada cenário são: endereços IP e MAC, pois cada máquina possui endereços distintos; usuário com privilégios administrativos e a respectiva senha, e; valores de MTTFs e MTTRs de acordo com o cenário.

Os experimentos foram conduzidos levando em consideração valores de MTTF e MTTR reais obtidos em (Dantas *et al.*, 2012; Hu *et al.*, 2010; Kim *et al.*, 2009). Para a realização dos experimentos aplicou-se o fator de redução 1000 nos valores de MTTF e MTTR reais. A aplicação do fator de redução permitiu executar experimentos mais rapidamente visto que, se os valores reais obtidos fossem adotados levariam semanas até que uma falha ocorresse e horas para que falhas fossem corrigidas, demandando muito tempo para execução de cada experimento. Os cenários 1b e 2b utilizam valores de MTTR 20% maiores que os valores adotados nos cenários 1a e 2a. O parâmetro forma (*shape*), k, para a distribuição Erlang presentes nos cenários 2a e 2b foi definido como k=2. As Tabelas 6.9 e 6.10 mostram os valores de MTTF e MTTR reais e os valores utilizados como parâmetros para a ferramenta já com o fator de redução aplicado e o acréscimo de 20% nos valores de MTTR dos cenários 1b e 2b.

Tabela 6.9 Parâmetros dos cenários 1a e 2a

Tipo de Componente	Real		Experimento	
	MTTF	MTTR	MTTF	MTTR
Hardware	8760 h	8 h	31536 s	28,8 s
Software	788 h	4 h	2836,8 s	14,4 s

As configurações estabelecidas para os cenários ajudam a determinar qual seria o impacto na disponibilidade do sistema caso a atividade de reparo seja retardada devido à falta de recursos.

Tabela 6.10 Parâmetros dos cenários 1b e 2b

Tipo de Componente	Real		Experimento	
	MTTF	MTTR	MTTF	MTTR
Hardware	8760 h	8 h	31536 s	34,5 s
Software	788 h	4 h	2836,8 s	17,3 s

6.2.3 Estudo de caso - Injeção de Falhas e Reparos em Ambientes de Nuvem Baseados em Distribuições de Probabilidade

Com os tempos de falhas e reparos de cada cenário determinado, conforme exposto nas Tabelas 6.11 e 6.12, e o tempo para execução de experimentos já estipulado, calculou-se a disponibilidade de cada cenário antes que os experimentos fossem realizados. Esta etapa é fundamental, pois o objetivo do teste com o EucaBomber é emular o comportamento especificado pela distribuição teórica ou empírica adotada no ferramental.

De conhecimento dos valores dos MTTFs e MTTRs estipulados foi possível obter a disponibilidade aproximada de cada cenário através da 2.1 e a indisponibilidade através da 2.2.

O cálculo da disponibilidade aproximada para os cenários 1a e 2a resultou em **96,56**%, enquanto que, para cenários 1b e 2b resultou em **95,89**%. A disponibilidade obtida através do cálculo para os cenários 1a e 2a são iguais visto que estes possuem os mesmos valores de MTTF e MTTR. A mesma observação pode ser feita quanto a igualdade da disponibilidade dos cenários 1b e 2b. A diferença entre a disponibilidade dos cenários 1a e 2a e 1b e 2b ocorre devido ao MTTR presente nos cenários 1b e 2b ser acrescido em 20%, em relação ao MTTR estabelecido para os cenários 1a e 2a.

Após o cálculo da disponibilidade para cada cenário, foram realizados os experimentos utilizando o EucaBomber. Os experimentos foram executados conforme descrito na

subseção 6.1.1 e alguns dos resultados obtidos estão na Tabela 6.11. Conforme é possível observar a disponibilidade obtida para os cenários 1a e 2a e 1b e 2b através do cálculo feito previamente e a disponibilidade obtida após o experimento com o EucaBomber, apresentada na Tabela 6.11, são relativamente próximos. É importante destacar que a disponibilidade alcançada através dos experimentos não corresponde exatamente ao estimado pelo cálculo devido aos tempos gerados pela ferramenta para realização dos testes serem aproximados aos valores de MTTF e MTTR informados.

Os resultados da Tabela 6.11 demonstram, como esperado, que a disponibilidade do cenário 1b diminui em relação ao 1a devido ao aumento do MTTR. Os experimentos também indicam que o aumento do MTTR tem um efeito maior sobre a indisponibilidade do *software* do que a indisponibilidade do *hardware*. Este comportamento ocorre tanto nos cenários baseados em distribuição exponencial quanto nos baseados na distribuição Erlang, pois, neste ambiente, as falhas de *software* são mais numerosas e frequentes que as falhas de *hardware*.

Tabela 6.11 Resultados da disponibilidade de todos os cenários

Métricas	**Cenários**			
	1a	**1b**	**2a**	**2b**
Disponibilidade (%)	94,4213	93,1626	95,4167	94,8438
Indisponibilidade (*hardware*) (%)	1,1892	1,3194	1,1140	1,1574
Indisponibilidade (*software*) (%)	4,3895	5,5179	3,4693	3,9988
Indisponibilidade (total) (%)	5,5787	6,8374	4,5833	5,1563

Os resultados mostrados na Tabela 6.12 foram obtidos através do monitoramento dos períodos de atividade e inatividade dos serviços. É possível observar que a diferença entre os cenários 1a e 1b é de cerca de 11%, quando se considera a inatividade do *hardware* (34,2 minutos em 1a em contraste com 38 minutos em 1b, considerando o período de 48 horas). Por outro lado, quando a inatividade de *software* é considerada, a diferença é de cerca de 25% (126,4 minutos em 1a contra 158,9 minutos em 1b, considerando o período de 48 horas). Pode-se chegar a conclusões semelhantes quando se observa os resultados dos cenários 2a e 2b, na Tabela 6.12. O tempo de falha de *hardware* aumenta apenas 3,7% de 2a para 2b, enquanto a inatividade de *software* aumenta 15,2%.

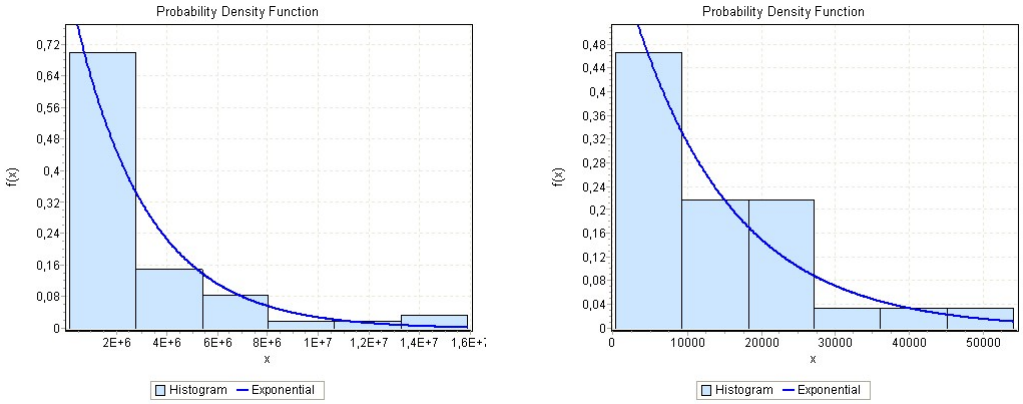
É importante destacar também a diferença observada entre os cenários 1a e 1b, onde o montante total da inatividade aumenta mais do que 30 minutos. Os resultados calculados da disponibilidade e os respectivos resultados obtidos através do experimento com o EucaBomber são muito próximos. A discrepância observada é atribuída a aleatoriedade dos tempos de falhas e reparos gerados pela ferramenta, o tempo necessário para ativar

processos que vieram a falhar e, por fim, o tempo que a máquina física leva para ser religada e estar apta para utilização. A baixa disponibilidade ([96,56; 93,1626]) é devida ao modo de falha considerado, pois neste estudo a falha de um único componente provoca a indisponibilidade do sistema.

Tabela 6.12 Tempo de atividade/inatividade de todos os cenários

Métricas	**Cenários**			
	1a	**1b**	**2a**	**2b**
Tempo de atividade (horas)	45,3	44,7	45,8	45,5
Tempo inativo (*hardware*) (min.)	34,2	38	32,1	33,3
Tempo inativo (*software*) (min.)	126,4	158,9	99,9	115,1
Tempo inativo (total) (min.)	160,6	196,9	132	148,5

Durante execução de cada cenário o EucaBomber coletou e armazenou informações em um arquivo no formato CSV dados como: o endereço IP da máquina na qual o evento de falha foi injetado e o respectivo reparo, se o evento é uma falha ou reparo, qual componente foi afetado, o instante em que ocorreu o evento. Com estas informações foi possível verificar se os tempos gerados para falhas e reparos de fato coincidiam com as distribuições selecionadas e parâmetros informados. As Figura 6.12 mostra, para o cenário 1a, o ajuste dos tempos de falha e reparo à distribuição exponencial. O teste de Kolmogorov-Smirnov (K-S) (com 95% de confiança), cujos valores são 0,0944 para os eventos de falha e 0,0882 para os eventos de reparo. Para ambos os casos, o índice é próximo de zero, portanto os dados mostram um bom ajuste à distribuição exponencial.



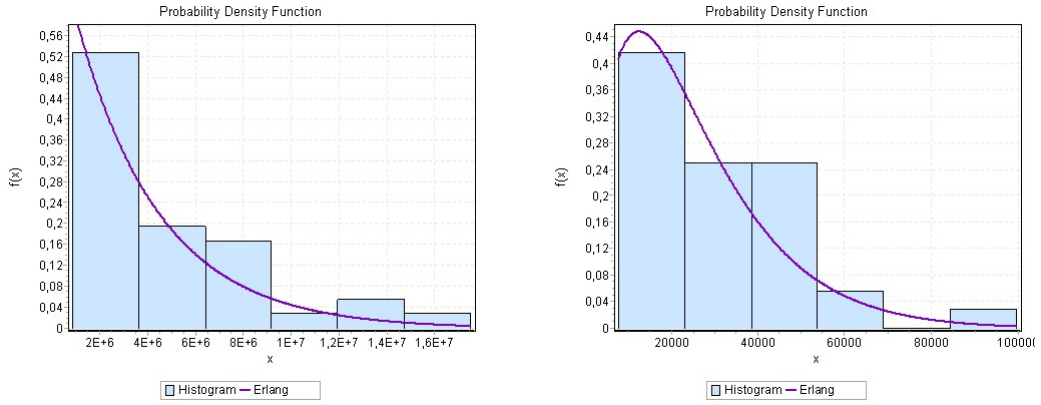
(a) Função densidade de probabilidade de eventos de falhas

(b) Função densidade de probabilidade de reparo

Figura 6.12 Distribuição de eventos no cenário 1a

A Figura 6.13 mostra o ajuste dos tempos de falha e reparo à distribuição Erlang. O

teste de K-S (com 95% de confiança) indica um índice de 0,2528 para eventos de falhas e 0,1597 para reparos.

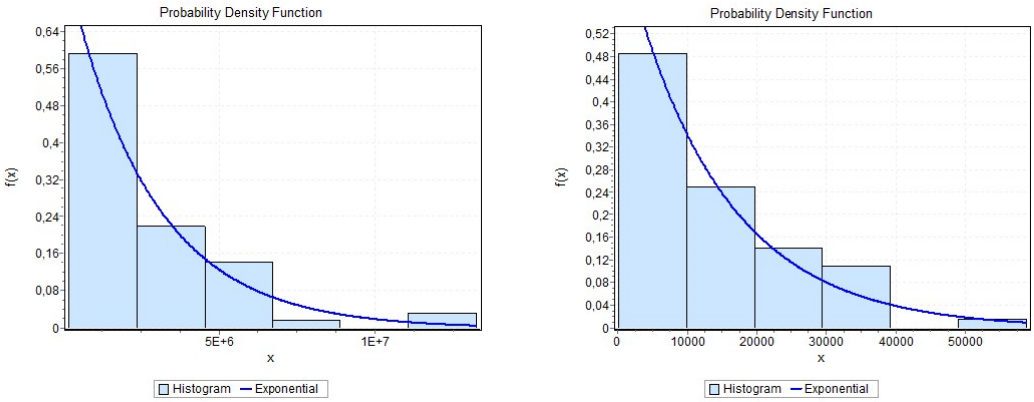


(a) Função densidade de probabilidade de eventos de falhas

(b) Função densidade de probabilidade de reparo

Figura 6.13 Distribuição de eventos no cenário 2a

A Figura 6.14 mostra o ajuste dos tempos de falha e reparo à distribuição exponencial, levando-se em consideração o cenário 1b. Como se pode observar a função densidade de ambos os eventos possui boa aproximação com a distribuição exponencial. O teste K-S (com 95% de confiança) resultou em 0,10804 para os eventos de falhas e 0,06619 para os reparos.



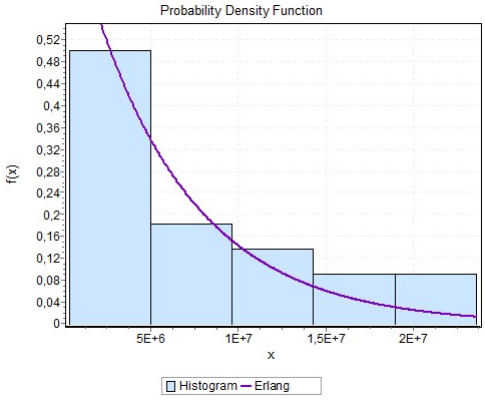
(a) Função densidade de probabilidade de eventos de falhas

(b) Função densidade de probabilidade de reparo

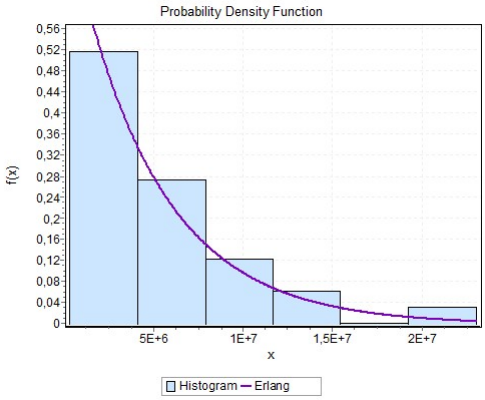
Figura 6.14 Distribuição de eventos no cenário 1b

O teste envolvendo os tempos de falhas e reparos do cenário 2b é apresentado na Figura 6.15. O teste K-S (com 95% de confiança) aponta um índice de 0,17533 para

eventos de falhas e 0,13041 para reparos respectivamente.



(a) Função densidade de probabilidade de eventos de falhas



(b) Função densidade de probabilidade de reparo

Figura 6.15 Distribuição de eventos no cenário 2b

Os resultados apresentados demonstram como o EucaBomber pode ser útil para administradores e analistas de sistemas, que necessitam avaliar a confiabilidade, disponibilidade de infraestrutura de nuvem e estratégias para melhoria da qualidade dos serviços suportados no ambiente.

6.3 Considerações Finais

Este capítulo apresentou dois estudos de caso envolvendo a utilização do WGSysEFT e do EucaBomber, duas ferramentas desenvolvidas a partir do FlexLoadGenerator. Os cenários elaborados em ambos os estudos de caso procuraram representar situações envolvendo carga de trabalho sintética para avaliar o desempenho sistema TEF e para injetar falhas em ambientes de nuvem objetivando avaliar o impacto destas falhas sobre a disponibilidade do sistema. Os resultados apresentados comprovam que ambas as ferramentas atenderam os objetivos para os quais foram construídas.

7

Conclusão e Trabalhos Futuros

Eu acredito demais na sorte. E tenho constatado que, quanto mais duro eu trabalho, mais sorte eu tenho.

—THOMAS JEFFERSON

Embora a criação de ferramentas de geração de eventos não seja a atividade principal durante o processo de concepção de um novo produto de *software*, é preciso levar em consideração que a necessidade de implementação desta ferramenta pode acarretar atrasos na entrega do sistema alvo. Uma das formas de minimizar esse transtorno é através do uso de *frameworks*.

Este trabalho propôs o desenvolvimento de um *framework*, intitulado FlexLoadGenerator, que pudesse servir de auxílio no processo de desenvolvimento de ferramentas geradoras de eventos sintéticos. O *framework* proposto oferece ao desenvolvedor um conjunto de métodos que se encarregam da (1) comunicação entre o gerador de evento e o sistema alvo, (2) criação e (3) gerenciamento de eventos, onde, o programador necessita apenas especificar qual evento deve ser criado e escolher como este deve se comportar. A flexibilidade provida pelo *framework* é uma de suas principais vantagens, onde o desenvolvedor opta por um dos meios disponibilizados para criação e gerenciamento de eventos. Caso não seja de interesse fazer uso da estrutura sugerida, pode-se utilizar (4) a biblioteca de geração de números aleatórios disponibilizada pelo FlexLoadGenerator de forma independente.

Para o desenvolvimento do *framework*, inicialmente, buscou-se na literatura o conhecimento necessário de como desenvolver um *framework*, seus pontos positivos e negativos, principais diferenças entre *frameworks* e bibliotecas, meios de implementação, flexibilização através de *hotspots* e formas de validação. Na sequência, trabalhos que

envolvessem a criação de ferramentas geradoras de eventos foram pesquisados. A partir da observação da forma de funcionamento e interação com os sistemas alvos em questão se pôde observar um conjunto de funcionalidades em comum, presentes em vários trabalhos, que, por fim, foram selecionadas, implementadas, testadas e disponibilizadas através do *framework*. Após a conclusão desta etapa, duas ferramentas intituladas WGSysEFT (*Workload Generator for Electronic Funds Transfer System*) e EucaBomber foram construídas como meios de validar o *framework* proposto neste trabalho. A primeira ferramenta visa auxiliar em testes de avaliação de desempenho de sistemas TEF e a segunda é voltada a experimentos de confiabilidade e disponibilidade em ambientes de nuvens gerenciados pelo Eucalyptus.

Durante o processo de implementação das ferramentas observou-se que o tempo e o esforço empregado na codificação foi reduzido devido a redução na criação de classes e métodos necessários para desenvolver as ferramentas, visto que em ambas as aplicações foi utilizado a estrutura fornecida pelo *framework*. Na etapa seguinte, foram realizados estudos de caso com intuito de demonstrar a eficiência das ferramentas implementadas com o auxílio do *framework*. O primeiro estudo de caso teve por foco avaliar o desempenho de máquinas de diferentes configurações, contendo a solução SCOPE para sistema TEF, sob influência de carga de trabalho. Enquanto o segundo estudo de caso visou avaliar a disponibilidade de um ambiente em nuvem mediante a ocorrência de falhas. Os resultados obtidos nos estudos de caso comprovam a eficiência das ferramentas criadas com o auxílio do FlexLoadGenerator aplicadas em suas áreas de atuação.

7.1 Contribuições, Limitações e Dificuldades

Entre as principais contribuições deste trabalho pode-se destacar:

- Desenvolvimento de um *framework* que objetiva contribuir para criação de geradores de eventos sintéticos. A validação do *framework* proposto foi realizada através desenvolvimento de duas aplicações construídas com seu auxílio, no qual se pode observar a redução de tempo e esforço empregado na codificação das ferramentas;

- Desenvolvimento de uma ferramenta de geração de carga de trabalho sintética com intuito de contribuir para estudos de avaliação de desempenho para sistemas TEF. A ferramenta proposta gera carga através de requisições simultâneas para realização de transações de crédito e débito, nos tipos à vista ou parcelado, onde a ferramenta realiza todas as trocas de mensagens necessárias para que transações sejam concluídas com sucesso. A ferramenta em questão pode ainda auxiliar no

planejamento de capacidade para estes sistemas;

- Desenvolvimento de uma ferramenta para estudos de dependabilidade, mais precisamente confiabilidade e disponibilidade, em ambientes de nuvem que sejam gerenciadas pela plataforma Eucalyptus. O *software* proposto atua inserindo e reparando falhas de acordo com tempos gerados baseados em distribuições de probabilidade;

Alguns resultados obtidos ao longo deste trabalho foram:

- Publicação do artigo "*A Tool for Automatic Dependability Test in Eucalyptus Cloud Computing Infrastructures*" na revista Computer and Information Science (Souza *et al.*, 2013);
- Aceitação do artigo "*EucaBomber: Experimental Evaluation of Availability in Eucalyptus Private Clouds*" no IEEE International Conference on Systems, Man, and Cybernetics (IEEE SMC 2013). Manchester, United Kingdom.

Algumas limitações e dificuldades foram observadas ao longo da construção deste trabalho. As mais relevantes foram:

- Determinar quais as funcionalidades que o *framework* deveria oferecer;
- Dificuldade em encontra pontos de flexibilidade (*hotspots*) para construção do *framework*;
- Pouca descrição em trabalhos presentes na literatura a respeito dos meios de concepção de ferramentas geradoras de eventos;
- Escassez de trabalhos com foco em avaliação de desempenho de sistemas TEF;
- Realização do estudo de caso utilizando a solução TEF intitulada SCOPE, devido ao modo de demonstração disponibilizar apenas a instalação conjunta de seus componentes (gerenciador de transações TEF, banco de dados e emulador de aprovação de transações). Esta condição imposta pelo SCOPE impede que estudos de caso mais elaborados sejam realizados como, por exemplo, o estudo de comportamento de um servidor dedicado apenas ao banco de dados responsável por armazenar transações concluídas pelo sistema;

- Realização do estudo de caso para verificar a disponibilidade de ambiente em nuvem gerenciado pela plataforma Eucalyptus. Neste caso, a limitação ocorreu quando desejou-se estudar o comportamento do ambiente de nuvem em teste com alta frequência de falhas e reparos durante um longo período de tempo. Durante este estudo percebeu-se que a versão do Eucalyptus utilizada nos testes apresentava problemas quando uma grande quantidade de falhas era injetada e reparadas em um curto espaço de tempo.

7.2 Trabalhos Futuros

Durante a concepção desta pesquisa foram observados alguns pontos que podem resultar em outros trabalhos que poderão dar-lhe continuidade. Este trabalho teve por foco o desenvolvimento de um *framework* que pudesse ser utilizado para minimizar o esforço na codificação de ferramentas geradoras de eventos e a validação deste através da criação de dois ferramentais para elaboração de estudos de caso. No entanto, o *framework* possui apenas alguns dos métodos que podem ser empregados para criação e gerenciamento de eventos.

Como trabalho futuro pode-se implementar outros métodos e incorporá-los ao *framework*, fazendo com que este possa oferecer uma quantidade maior de possibilidades ao desenvolvedor, do que os disponibilizados na atualidade. Embora o *framework* tenha sido projetado para auxiliar na criação de ferramentas com intuito de ser útil em questões de avaliação de desempenho e dependabilidade, os mecanismos fornecidos pelo FlexLoadGenerator podem ser reaproveitados para apoiar a implementação de aplicações com propósitos diferentes dos supracitados.

A biblioteca de geração de números aleatórios baseados em distribuição de probabilidade também pode ser ampliada, onde um novo conjunto de distribuições pode ser incorporado, aumentando assim o número de possibilidades para geração de números aleatórios.

Além disso, as ferramentas concebidas por meio do FlexLoadGenerator podem ser expandidas com a adição de novas funcionalidades. Para o WGSysEFT pode-se adicionar funcionalidades que abranjam pagamentos com cheques e vale refeição, ou ainda, recarga de telefones móveis, visto que estas opções estão também disponíveis em sistemas TEF atuais. Quanto ao EucaBomber, este também pode ser estendido para abranger outros ambientes de nuvem, além da plataforma Eucalyptus. Falhas mais elaboradas de *hardware* e *software* podem ser incorporadas sem grandes esforços.

Referências

ABECS (2013). http://www.abecs.org.br. Último acesso em Maio de 2013.

Amazon (2013a). Amazon elastic compute cloud - ec2. http://aws.amazon.com/ec2. Último acesso em Junho de 2013.

Amazon (2013b). Amazon simple storage service. http://aws.amazon.com/pt/s3. Último acesso em Junho de 2013.

Amazon (2013c). Instâncias da amazon ec2. http://aws.amazon.com/pt/ec2/instance-types. Último acesso em Junho de 2013.

Anderson, K., Bigus, J., Bouillet, E., Dube, P., Halim, N., Liu, Z., and Pendarakis, D. (2006). Sword: Scalable and flexible workload generator for distributed data processing systems. In *Simulation Conference, 2006. WSC 06. Proceedings of the Winter*, pages 2109–2116.

Avizienis, A., Laprie, J., and Randell, B. (2001). Fundamental concepts of dependability.

Bahga, A. and Madisetti, V. K. (2011). Synthetic workload generation for cloud computing applications. *JSEA*, **4**(7), 396–410.

Baraza, J.-C., Gracia, J., Gil, D., and Gil, P. (2000). A prototype of a vhdl-based fault injection tool. In *Defect and Fault Tolerance in VLSI Systems, 2000. Proceedings. IEEE International Symposium on*, pages 396–404.

Borko Furht, A. E., editor (2010). *Handbook of Cloud Computing*. Springer, 2010.

Bradesco (2013). http://www.bradescocartoes.com.br. Último acesso em Janeiro de 2013.

Busari, M. and Williamson, C. (2002). Prowgen: A synthetic workload generation tool for simulation evaluation of web proxy caches. *Computer Networks*, **38**, 779–794.

Cao, R., Chen, Y., and Kang, R. (2012). Critical review of system failure behavior analysis method. In *Prognostics and System Health Management (PHM), 2012 IEEE Conference on*, pages 1–10.

Carneiro, C. (2003). *Frameworks de Aplicações Orientadas a Objetos - Uma Abordagem Iterativa e Incremental*. Dissertação, UNIFACS.

Cielo (2013). http://www.cielo.com.br. Último acesso em Janeiro de 2013.

D, J., Murari, K., Raju, M., RB, S., and Girikumar, Y. (2010). *Eucalyptus Beginner's Guide*, uec edition.

Dantas, J. R., Matos, R., Araujo, J., and Maciel, P. (2012). An availability model for eucalyptus platform: An analysis of warm-standy replication mechanism. In *The 2012 IEEE Int. Conf. on Systems, Man, and Cybernetics (IEEE SMC 2012)*, Seoul.

Deitel, H. and Deitel, P. (2010). *Java: Como Programar*. How to program series. Pearson Prentice Hall.

Denneulin, Y., Romagnoli, E., and Trystram, D. (2004). A synthetic workload generator for cluster computing. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 243–.

Devroye, L. (1986). Non-uniform random variate generation.

EBAPE / Tenório, F. (2004). *Responsabilidade Social Empresarial: Teoria E Prática*. Coleção FGV prática. FGV.

Ebeling, C. (2004). *An introduction to reliability and maintainability engineering*. Electrical engineering series. McGraw-Hill.

Eclipse (2013). http://www.eclipse.org/. Último acesso em Junho de 2013.

Ejlali, A., Miremadi, S.-G., Zarandi, H., Asadi, G., and Sarmadi, S. (2003). A hybrid fault injection approach based on simulation and emulation co-operation. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pages 479–488.

Eucalyptus (2009). *Eucalyptus Open-Source Cloud Computing Infrastructure - An Overview*. Eucalyptus Systems, Inc.

Eucalyptus (2013a). Eucalyptus - the open source cloud platform. http://open.eucalyptus.com. Último acesso em Abril de 2013.

Eucalyptus (2013b). Overview of eucatools. http://www.eucalyptus.com. Último acesso em Junho de 2013.

Fortier, P. J. and Michel, H. (2002). *Computer Systems Performance Evaluation and Prediction*. Butterworth-Heinemann, Newton, MA, USA.

Foster, I., Zhao, Y., Raicu, I., and Lu, S. (2008). Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10.

Fowler, M. (2004). *UML Distilled: A Brief Guide to the Standard Object Modeling Language ; [covers Through Version 2.0 OMG UML Standard]*. Object Technology Series. Addison Wesley Professional.

Friesenbichler, W., Panhofer, T., and Steininger, A. (2010). A deterministic approach for hardware fault injection in asynchronous QDI logic. In *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th Int. Symp. on*, pages 317 –322.

Fujita, H., Matsuno, Y., Hanawa, T., Sato, M., Kato, S., and Ishikawa, Y. (2012). Ds-bench toolset: Tools for dependability benchmarking with simulation and assurance. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–8.

Galindo, H., Santos, W., Maciel, P., Silva, B., Galdino, S., and Pires, J. (2009). Synthetic workload generation for capacity planning of virtual server environments. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE Int. Conf. on*, pages 2837 –2842.

Galindo, H., Guedes, E., Maciel, P., Silva, B., and Galdino, S. (2010). Wgcap: A synthetic trace generation tool for capacity planning of virtual server environments. In *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*, pages 2094–2101.

Ganger, G. R. (1995). Generating representative synthetic workloads: An unsolved problem. In *in Proceedings of the Computer Measurement Group (CMG) Conference*, pages 1263–1269.

Google (2013a). Google app engine. https://appengine.google.com/start. Último acesso em Junho de 2013.

Google (2013b). Google compute engine. https://cloud.google.com. Último acesso em Junho de 2013.

Google (2013c). Google docs. https://docs.google.com. Último acesso em Junho de 2013.

Hayter, A. J. (2007). *Probability & Statistics For Engineers & Scientists, 8/E*. Pearson Education.

httperf (2013). http://www.hpl.hp.com/research/linux/httperf/. Último acesso em Fevereiro de 2013.

Hu, T., Guo, M., Guo, S., Ozaki, H., Zheng, L., Ota, K., and Dong, M. (2010). Mttf of composite web services. In *Parallel and Distributed Processing with Applications (ISPA), 2010 Int. Symp. on*, pages 130 –137.

ISO (2013). Iso 8583-1:2003. financial transaction card originated messages. interchange message specifications. http://www.iso.org. Último acesso em Novembro de 2012.

Itaú (2013). http://www.itau.com.br. Último acesso em Janeiro de 2013.

Itautec (2013a). Guia de referência para o scope - solução completa para pagamento eletrônico.

Itautec (2013b). Manual do integrador (desenvolvedor) - solução completa para pagamento eletrônico.

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley Professional Computing. Wiley.

Java (2013). http://www.oracle.com/. Último acesso em Maio de 2013.

Jeitler, M., Delvai, M., and Reichor, S. (2009). Fuse - a hardware accelerated hdl fault injection tool. In *Programmable Logic, 2009. SPL. 5th Southern Conference on*, pages 89–94.

JMeter (2013). http://jmeter.apache.org. Último acesso em Agosto de 2013.

Kant, K., Tewari, V., and Iyer, R. (2001). Geist: a generator for e-commerce internet server traffic. In *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, pages 49–56.

Kim, D., Machida, F., and Trivedi, K. (2009). Availability modeling and analysis of a virtualized system. In *Dependable Computing, 2009. PRDC '09. 15th IEEE Pacific Rim Int. Symp. on*, pages 365–371.

Krishnamurthy, D., Rolia, J., and Majumdar, S. (2006). A synthetic workload generation technique for stress testing session-based systems. *Software Engineering, IEEE Transactions on*, **32**(11), 868–882.

Lilja, D. J. (2000). *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, New York.

Looker, N., Munro, M., and Xu, J. (2004). Ws-fit: a tool for dependability analysis of web services. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 2, pages 120–123 vol.2.

Mansour, M., Wolf, M., and Schwan, K. (2004). Streamgen: a workload generation tool for distributed information flow applications. In *Parallel Processing, 2004. ICPP 2004. International Conference on*, pages 55–62 vol.1.

Mattsson, M. (1996). Object-oriented frameworks - a survey of methodological issues.

Mattsson, M. and Bosch, J. (1997). Framework composition: problems, causes and solutions. In *Technology of Object-Oriented Languages and Systems, 1997. TOOLS 23. Proceedings*, pages 203–214.

Mell, P. and Grance, T. (2011). The nist definition of cloud computing (draft). nist special publication.

Neamtiu, I. and Dumitras, T. (2011). Cloud software upgrades: Challenges and opportunities. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2011 International Workshop on the*, pages 1–10.

Obaidat, M. S. and Boudriga, N. A. (2010). *Fundamentals of Performance Evaluation of Computer and Telecommunications Systems*. Wiley-Interscience, New York, NY, USA.

Panda, A., Avakian, A., and Vemuri, R. (2011). Configurable workload generators for multicore architectures. In *SOC Conference (SOCC), 2011 IEEE International*, pages 179–184.

Patil, P., Kulkarni, P., and Bellur, U. (2011). Virtperf: A performance profiling tool for virtualized environments. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 57–64.

Penczek, L. (2008). *AFR: Uma Abordagem para a Sistematização do Reúso de Frameworks Orientados a Aspectos*. Dissertação, PUC/RS.

Perfmon (2013). Monitor de desempenho. http://technet.microsoft.com/en-us/library/bb490957.aspx. Último acesso em Dezembro de 2012.

Popa, M. and Slavici, T. (2009). Embedded server with wake on lan function. In *EUROCON 2009, EUROCON '09. IEEE*, pages 365 –370.

Pradhan, D. K., editor (1996). *Fault-tolerant computer system design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

REDECARD (2013). http://www.redecard.com.br/. Último acesso em Janeiro de 2013.

Rimal, B., Choi, E., and Lumb, I. (2009). A taxonomy and survey of cloud computing systems. In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 44–51.

Salesforce (2013). http://www.salesforce.com. Último acesso em Junho de 2013.

Schneider, S., Mueller, U., and Tiegelbekkers, D. (2005). A reactive workload generation framework for simulation-based performance engineering of system interconnects. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, pages 484–487.

Sebesta, R. (2003). *Conceitos de Linguagens de Programacao*. BOOKMAN COMPANHIA ED.

Sommerville, I., Melnikoff, S., Arakaki, R., and Andrade Barbosa, E. (2008). *Engenharia de software*. ADDISON WESLEY BRA.

Sousa, E., Maciel, P., Medeiros, E., Souza, D., Lins, F., and Tavares, E. (2012). Evaluating eucalyptus virtual machine instance types: A study considering distinct workload demand. pages 130–135. Copyright (c) IARIA, 2012, The Third Int. Conf. on Cloud Computing, GRIDs, and Virtualization.

Souza, D. S. L., Matos Junior, R. S., Araujo, J. C. T., Lira Neto, V. A., and Maciel, P. R. M. (2013). A tool for automatic dependability test in eucalyptus cloud computing infrastructures. *Computer and Information Science*.

Svenningsson, R., Eriksson, H., Vinter, J., and Torngren, M. (2010). Model-implemented fault injection for hardware fault simulation. In *Model-Driven Engineering, Verification, and Validation (MoDeVVa), 2010 Workshop on*, pages 31–36.

Tomhave, B. L. (2005). Alphabet soup: Making sense of models, frameworks, and methodologies. *secure consulting, Available. at< www. secureconsulting. net/Papers/Alphabet_Soup. pdf>[Accessed 20 March 2011]*.

Trivedi, K., Kim, D., Roy, A., and Medhi, D. (2009). Dependability and security models. In *Design of Reliable Communication Networks, 2009. DRCN 2009. 7th International Workshop on*, pages 11–20.

Vacaro, J. and Weber, T. (2006). *Injeção de Falhas na Fase de Teste de Aplicações Distribuídas*. XX Simpósio Brasileiro de Engenharia de Software, Florianópolis.

Van Ertvelde, L. and Eeckhout, L. (2010). Workload reduction and generation techniques. *Micro, IEEE*, **30**(6), 57–65.

Wanner, P. (2003). *Ferramenta de Injeção de Falhas para Avaliação de Segurança*. Dissertação, Universidade Federal do Rio Grande do Sul. Instituto de Informática. Programa de Pós-Graduação em Computação.

Weber, T. (2002). *Um roteiro para exploração dos conceitos básicos de tolerância a falhas*. Instituto de Informática, UFRGS, Porto Alegre.

Weinkopf, J., Harbich, K., and Barke, E. (2006). Parsifal: A generic and configurable fault emulation environment with non-classical fault models. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6.

Zhang, H. (2010). Research about software fault injection technology based on distributed system. In *Machine Vision and Human-Machine Interface (MVHI), 2010 International Conference on*, pages 518–521.

Zhang, T., Xiao, X., and Qian, L. (2008). Modeling object-oriented framework with z. In *Computer Science and Computational Technology, 2008. ISCSCT '08. International Symposium on*, volume 2, pages 165–170.

Zhang, Y., Liu, B., and Zhou, Q. (2011). A dynamic software binary fault injection system for real-time embedded software. In *Reliability, Maintainability and Safety (ICRMS), 2011 9th Int. Conf. on*, pages 676 –680.

Zhou, Z., Tang, W., Zheng, Z., and Lan, Z. e Desai, N. (2011). Evaluating performance impacts of delayed failure repairing on large-scale systems. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 532–536.

Ziade, H., Ayoubi, R., and Velazco, R. (2004). A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, **1**(2), 171–186.

Appendices

A

Diagrama de Classe

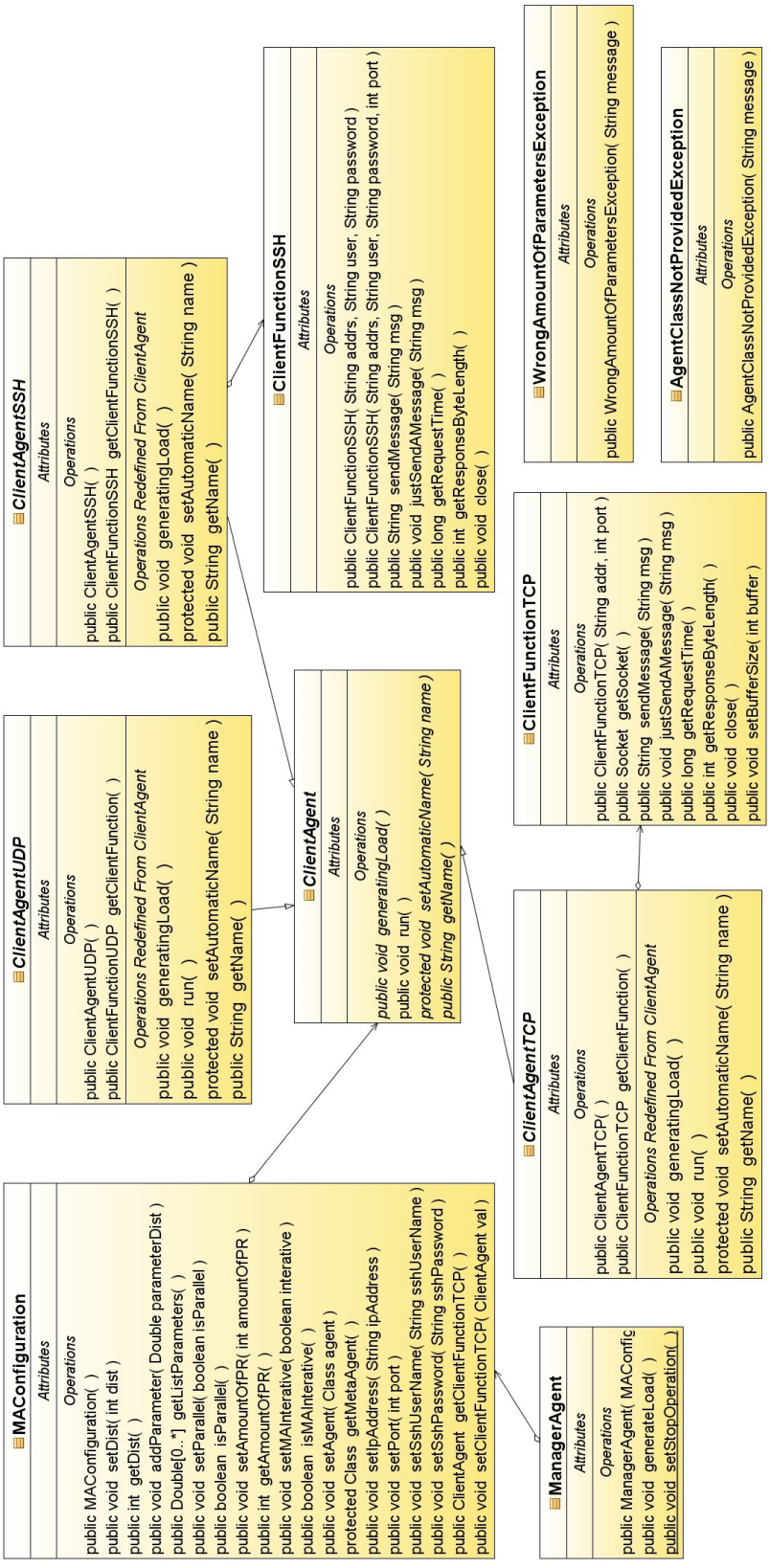


Figura A.1 Diagrama de Classe FlexLoadGenerator

B

Descrição de Classes e Métodos FlexLoadGenerator

O FlexLoadGenerator é composto por onze classes no total. Nove das onze classes se encarregam de estabelecer a comunicação com o sistema em teste, criar e gerenciar eventos sintéticos. As duas classes restantes são responsáveis por tratamento interno de exceções caso o *framework* seja utilizado de modo incorreto.

B.1 Comunicação

Sete, das onze classes, que formam o *FlexLoadGenerator* são incumbidas a estabelecer comunicação, são elas: *ClientAgent*, *ClientAgentTCP*, *ClientAgentUDP*, *ClientAgentUDP*, *ClientFunctionTCP*, *ClientFunctionUDP* e *ClientFunctionSSH*.

ClientAgent: é a superclasse de *ClientAgentTCP*, *ClientAgentUDP* e *ClientAgentSSH*. De caráter abstrato esta classe contém os seguintes métodos: *generatingLoad()*, *setAutomaticName(String name)* e *getName()*. Estes métodos serão discutidos em maiores detalhes nas subclasses de *ClientAgent*.

ClientAgentTCP: classe abstrata que tem por objetivo representar a unidade de geração de eventos para o protocolo TCP. Os construtores e métodos disponibilizados por essa classe são:

- ***ClientAgentTCP()*:** este construtor é utilizado para criar um objeto do tipo *ClientFunctionTCP*, e automaticamente elabora a conexão com o sistema em estudo;
- ***generatingLoad()*:** método abstrato advindo da classe *ClientAgent*. Este método deve ser sobrescrito pelo desenvolvedor e no seu interior obrigatoriamente deve conter quais eventos devem ser criados;

- ***getClientFunction()*:** este método retorna o *clientFunction* associado que será utilizado para enviar mensagens ao sistema alvo;

- ***getName()*:** este método retorna o nome do objeto responsável por gerar eventos após este já ter sido criado; e

- ***setAutomaticName(String name)*:** este método é responsável por atribuir nomes aos objetos responsáveis por criar eventos. Este método não possui retorno.

Da mesma forma que *ClientAgentTCP* é uma subclasse abstrata de *ClientAgent*, as classes *ClientAgentUDP* e *ClientAgentSSH* também são, possuem os mesmos métodos e realizam as mesmas tarefas apenas substituindo o protocolo TCP por UDP, ou SSH2, respectivamente. Portanto, a descrição anterior dos métodos de *ClientAgentTCP* é aplicável aos métodos pertencentes as classes *ClientAgentUDP* e *ClientAgentSSH*.

***ClientFunctionTCP*:** esta classe é responsável pela comunicação entre a ferramenta injetora de eventos a ser implementada e o sistema alvo. Para que esta classe possa ser utilizada é necessário que o sistema alvo seja acessado através do protocolo TCP. Os seguintes métodos são disponibilizados:

- ***ClientFunctionTCP(String addr, int port)*:** este construtor é responsável por estabelecer a conexão entre o gerador de eventos e o sistema alvo através do protocolo TCP. Recebe como parâmetro o endereço e porta do servidor de destino para estabelecer a comunicação. O endereço recebido por este parâmetro foi definido como do tipo *String*, pois o usuário pode informar tanto o endereço IP (*Internet Protocol*) da máquina como o nome de domínio que deverá ser transformado em endereço IP por um servidor DNS (*Domain Name System*);

- ***sendMessage(String msg)*:** este método é responsável por enviar mensagens e receber a respectiva resposta do servidor;

- ***justSendAMessage(String msg)*:** diferentemente do método *sendMessage(String msg)*, este método apenas faz o envio da mensagem, mas não espera pela resposta do servidor;

- ***setBufferSize(int buffer)*:** este método possibilita que o usuário controle o tamanho do *buffer* que armazena mensagens recebidas. Por padrão, o tamanho do *buffer* é de 2048 *bytes*;

- ***getRequestTime()*:** este método retorna o tempo decorrido desde o último envio de mensagem. Deve ser chamado após a chamada do método *sendMessage(String msg)* ou do método *justSendAMessage(String msg)*;

- ***getResponseByteLength()*:** retorna a quantidade de *bytes* enviados pela última resposta dada pelo servidor. Deve ser utilizado apenas após a chamada do método *sendMessage(String msg)*; e

- ***close()*:** este método é responsável por encerrar a comunicação entre a ferramenta injetora de eventos e o sistema em teste.

Os métodos *getRequestTime()* e *getResponseByteLength()* foram desenvolvidos para facilitar a elaboração de relatórios.

Com a mesma finalidade da classe *ClientFunctionTCP*, as classes *ClientFunctionUDP* e *ClientFunctionSSH* se diferenciam apenas devido aos protocolos de comunicação UDP e SSH2, respectivamente. Estas classes compartilham vários métodos, com as seguintes exceções:

- ***Classe ClientFunctionUDP*:** devido ao protocolo UDP não ser orientado a conexão, esta classe não oferece os métodos *sendMessage(String msg)*, *getRequestTime()* e *getResponseByteLength()*. O objetivo, neste caso, é o envio de dados seguidamente sem a necessidade de recebimento de respostas;

- ***Classe ClientFunctionSSH*:** esta classe, diferente das demais, apresenta dois construtores. O primeiro, *ClientFunctionSSH(String addr, String user, String password)*, possui parâmetros diferenciados devido ao protocolo SSH2 necessitar do endereço para o qual a mensagem deve ser enviada, o nome do usuário e a senha de acesso. Caso o usuário opte por usar este construtor, deve se certificar que a porta 22 é a disponibilizada pelo servidor para estabelecer a comunicação. Se, por algum motivo, o servidor fizer uso de outra porta que não a 22, o usuário deve utilizar o construtor *ClientFunctionSSH(String addrs, String user, String password, int port)*. Este construtor oferece ao desenvolvedor a possibilidade de configurar outra porta para estabelecer a comunicação.

B.2 Criação e Gerenciamento de Eventos

Duas, das onze classes presentes no FlexLoadGenerator, são responsáveis pela criação e gerenciamento de eventos, são elas: *MAConfiguration* e *ManagerAgent*.

***MAConfiguration*:** esta classe contém um conjunto de métodos usados para preparar e controlar o processo de geração de eventos. Através dela o usuário pode especificar como a geração deve ser feita e como deve ser a administração dos eventos ao longo da execução da ferramenta concebida com o auxílio do *framework*. Uma parte importante desta classe está relacionada às distribuições de probabilidade disponibilizadas através da biblioteca de geração de números aleatórios. Os métodos presentes nesta classe são:

- ***MAConfiguration()*:** construtor responsável por inicializar uma instância de *MAConfiguration*;
- ***setDist(int dist)*:** é através deste método que o usuário deve informar qual a distribuição de probabilidade escolhida. As distribuições devem ser informadas através de sua constante. Cada distribuição disponibilizada possui sua constante, onde o nome da constante é o nome da distribuição em inglês no formato caixa alta como, por exemplo, ERLANG. Neste caso, a única exceção a nomenclatura das constantes se dá apenas para utilização da distribuição empírica. O parâmetro recebido por *setDist(int dist)* é do tipo inteiro porque cada constante internamente está ligada a um número inteiro que faz referência a uma distribuição de probabilidade específica;
- ***getDist()*:** obtém o número referente à distribuição de probabilidade definida anteriormente;
- ***addParameter(Double parameterDist)*:** este é um método que recebe um valor do tipo *Double* e o adiciona em uma lista dos parâmetros da distribuição de probabilidade escolhida. O usuário deverá chamar esse método uma ou mais vezes, a depender da quantidade de parâmetros requerida pela distribuição escolhida, como por exemplo, se o utilizador escolher a distribuição Erlang, este deverá chamar o método duas vezes: a primeira para configurar a taxa e a segunda para configurar a forma da distribuição. Os dois primeiros valores da lista estão preenchidos por padrão com os valores 0.0 e 10000000.0, que representam o intervalo entre os números que a biblioteca pode gerar. Caso uma distribuição que possua mais de um parâmetro seja escolhida, é necessário observar na documentação do *framework* qual a ordem em que os parâmetros devem ser inseridos;
- ***getListParameters()*:** este método retorna a lista de parâmetros informados para a distribuição de probabilidade escolhida;

- ***setParallel(boolean isParallel)*:** método do tipo booleano que, por padrão, é configurado como *true*, ocasionando que a geração de eventos ocorra de forma paralela, ou seja, mais de um evento é disparado simultaneamente. O usuário pode reconfigurar este parâmetro para *false*. Neste caso, o disparo do eventos acontecerá de forma sequencial;

- ***isParallel()*:** este método retorna se a geração de eventos foi definida para ocorrer de forma paralela ou não;

- ***setAmountOfPR(int amountOfPR)*:** este método define a quantidade de eventos a serem criados em paralelo quando *setParallel(boolean isParallel)* é definido como *true*. No entanto, se *setParallel(boolean isParallel)* for configurado como *false* este método não deve ser utilizado;

- **getAmountOfPR():** este método retorna a quantidade de eventos a serem criados em paralelo, conforme definido em *setAmountOfPR()*;

- ***setRound(boolean round)*:** define se eventos ocorrerão em rodadas, ou seja, a cada rodada ocorre o disparo de novos eventos sem que os já em execução tenham chegado ao final. Por padrão, este atributo é definido como *true*. Quando *true*, o ferramental cria de tempos em tempos (em rodadas), de acordo com o valor resultante da distribuição e parâmetros informados, novos eventos sem esperar que os anteriormente concebidos terminem sua execução. Quando definido como *false*, este método gera apenas uma rodada de eventos;

- ***isRound()*:** retorna *true* se o eventos ocorrerem rodadas, e *false* caso contrário;

- ***setAgent(Class agent)*:** este método define a classe que será utilizada pata gerar os eventos;

- ***getMetaAgent()*:** retorna a classe definida pelo usuário;

- ***setIpAddress(String ipAddress)*:** este método recebe um parâmetro do tipo *String* contendo o endereço IP com o qual se deseja estabelecer comunicação. Este método é utilizado independente do protocolo de conexão escolhido;

- ***setPort(int port)*:** este método recebe um parâmetro do tipo inteiro contendo a porta para comunicação com o sistema alvo da geração de eventos. Este parâmetro também pode ser utilizado quando o protocolo SSH2 for escolhido, entretanto, o

usuário só deverá configura-lo quando necessitar utilizar outra porta, que não a 22 (definida como padrão);

- ***setSshUserName(String sshUserName)*:** este atributo recebe um *String* contendo o *login* do usuário. Só deve ser utilizado quando o protocolo SSH2 é escolhido;
- ***setSshPassword(String sshPassword)*:** este atributo recebe um *String* contendo a senha do usuário. Só deve ser utilizado quando o protocolo SSH2 é escolhido.

ManagerAgent: esta classe é responsável pelo gerenciamento da criação e atuação de eventos. Os métodos disponibilizados por esta classe são:

- ***ManagerAgent(MAConfiguration configuration)*:** este construtor recebe como parâmetro um objeto da classe *MAConfiguration* devidamente configurado;
- ***generateLoad()*:** método responsável por gerenciar os eventos criados de acordo com os parâmetros configurados na classe *MAConfiguration*;
- ***setStopOperation()*:** método responsável por interromper a execução de eventos criados.

C

Manual da Ferramenta WGSysEFT

A carga gerada pela aplicação depende das escolhas que são feitas no momento em que a primeira *interface* gráfica do WGsysEFT é preenchida. A Figura C.1 ilustra o *screenshot* referente a tela inicial do WGSysEFT. As opções ofertadas referentes ao tipo de cartão (crédito ou débito) e forma de pagamento (à vista ou parcelado) caracterizam a composição de possíveis cenários, através de combinações.

Inicialmente, é preciso optar por qual tipo de transação será realizada: crédito (*Credit Card*), débito (*Debit Card*), ou ainda ambas as operações (*Credit e Debit Card*). Após a escolha do tipo de transação, é preciso definir ao menos uma forma de pagamento, para cada tipo selecionado. O gerador dispõe de duas formas de pagamento: à vista (*Spot purchase*), e parcelado (*Forward purchase*). Mais uma vez, pode-se adotar uma das formas de pagamento ou ambas para cada tipo se selecionado.

Antes de selecionar quaisquer alternativas fornecidas pela ferramenta é preciso verificar junto ao sistema TEF, em teste, quais serviços estão cadastrados e disponíveis. Uma observação importante a ser feita é em relação a escolha de *Debit Card* juntamente com a opção *Forward Purchase*. Nem sempre a opção de pagamento parcelado em cartões de débito estará apta para execução de transações no sistema TEF, visto que é de escolha do estabelecimento comercial selecionar que tipo de transações serão aceitas.

À medida que o usuário seleciona os cartões (crédito e/ou débito) e formas de pagamento (a vista e/ou parcelado) a ferramenta habilita os campos correspondentes aos anteriormente selecionados, presentes no painel inferior da *interface* gráfica onde se lê "*Enter the percentage for execution of transactions*". Nestes campos o usuário deve informar o percentual para ocorrência de cada tipo de transação previamente selecionado. É através deste percentual que a ferramenta decide qual o tipo de transação deve ser realizada e com que frequência, por exemplo, 34% para transações de crédito à vista, 33% para crédito parcelado e 33% para débito à vista.

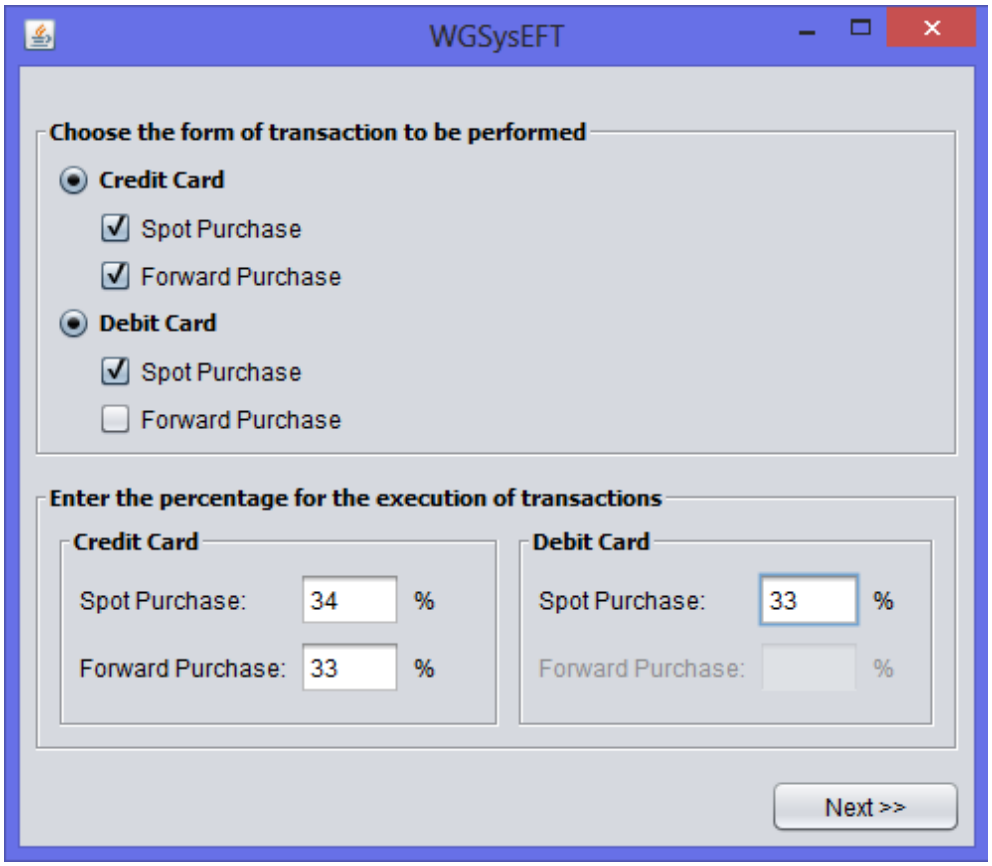


Figura C.1 *Screenshot* WGSysEFT (Tela inicial)

O uso do recurso de percentual possibilita direcionar a carga sintética. Por exemplo, ao longo de duas semanas o estabelecimento comercial *X* teve 34% das vendas realizadas com cartão de crédito à vista, 33% com cartão de crédito parcelado, 33% das vendas pagas com cartão de débito à vista. Através desse mecanismo, é possível recriar uma carga sintética que se aproxime da original.

Ao acionar o botão *Next*, presente na tela inicial, a segunda *interface* gráfica, pertencente ao WGSysEFT, é apresentada. Observe que a etapa anterior constrói o cenário para geração de carga. Enquanto a fase seguinte, limita-se à inserção de parâmetros necessários para controle da carga. O *screenshot* da segunda *interface* gráfica é exposto na Figura C.2.

O painel superior, presente na Figura C.2, onde se lê "*Choose the probability distribution*", remete a seleção de uma das distribuições de probabilidade disponibilizadas. O ferramental proposto dispõe das seguintes distribuições de probabilidade: Erlang, Exponencial, Geométrica, Lognormal, Normal, Pareto, Poisson, Triangular, Uniforme e Weibull.

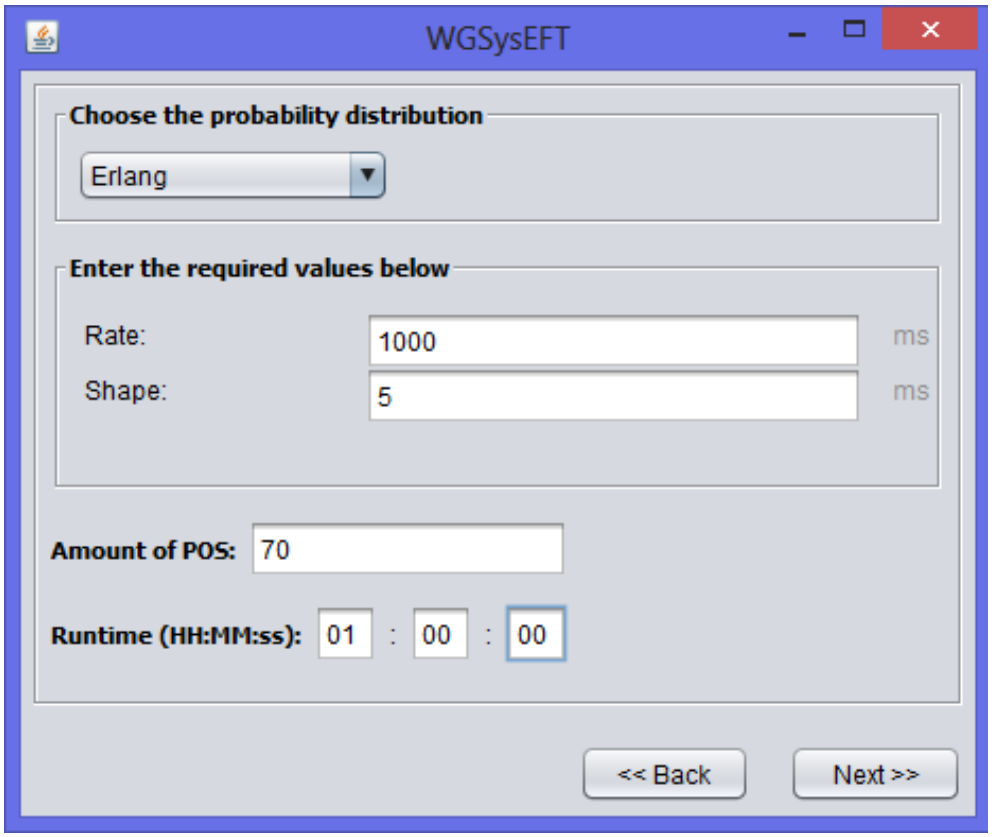


Figura C.2 *Screenshot* WGSysEFT (Segunda *interface* gráfica)

Após a escolha da distribuição, são apresentados os parâmetros que devem ser obrigatoriamente preenchidos. Cada distribuição tem parâmetros específicos que devem ser inseridos no(s) campo(s) apresentados no painel central. Por exemplo, a distribuição Triangular possui mínimo (*minimum*), máximo (*maximum*), e moda (*mode*) respectivamente. Caso a distribuição escolhida possua mais de um parâmetro, então a ferramenta exibe a quantidade de campos de texto de acordo com o número de parâmetros, e informa a ordem de preenchimento.

Os parâmetros das distribuições devem ser preenchidos em função de tempo. Como o objetivo da ferramenta é gerar carga de trabalho sintética para teste de desempenho do sistema TEF, então, a unidade de tempo utilizada corresponde a milissegundos. Os valores gerados a partir das distribuições são utilizados como intervalo de tempo entre o final da execução de uma transação e o início da próxima.

Logo após a escolha da distribuição, e configuração de seus parâmetros, é preciso informar a quantidade de PDVs que serão utilizados ao longo do experimento. Essa informação deve ser preenchida no campo de texto correspondente aos dizeres "*Amount*

os POS", onde *POS* significa Pontos de Venda (PDV).

Por fim, o usuário deve informar no campo onde se lê "*Runtime (HH:MM:ss)*", o tempo de execução da ferramenta, no formato estabelecido.

A segunda *interface* gráfica apresenta um botão "*« Back*" que retorna a tela inicial permitindo que se possa voltar a fase inicial e realizar alterações. O botão "*Next »*" aciona a terceira, e última, interface da ferramenta proposta. O *screenshot* da terceira tela do WGSysEFT é apresentada na Figura C.3.

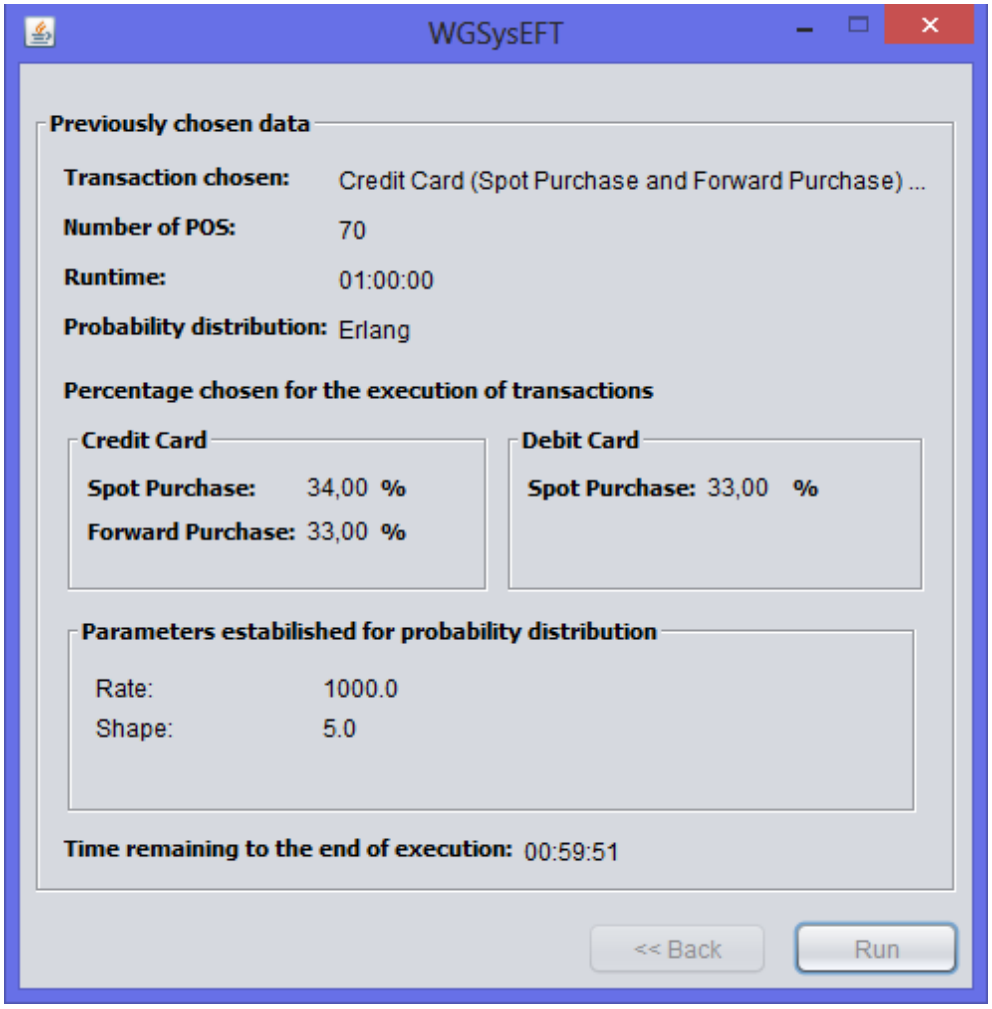


Figura C.3 *Screenshot* WGSysEFT (Terceira *interface* gráfica)

A interface gráfica, apresentada na Figura C.3, retrata as escolhas anteriores do usuário para devida averiguação se o que foi informado corresponde ao desejado. Esta *interface* oferece ao usuário a possibilidade de retornar a tela anterior para modificações através do botão "*« Back*". O botão "*Run*" aciona a ferramenta para geração de carga, conforme o cenário elaborado nas fases anteriores.

O acionamento da aplicação faz com que a contagem regressiva para o término de sua execução seja disparado, conforme mostra o a parte inferior do painel, indicada por "*Time remaining to the end of execution*". Ao final da contagem regressiva, a ferramenta conclui sua atuação.

Se a ferramenta não conseguir estabelecer comunicação com o sistema TEF, ou se transações não puderem ser realizadas devido à falta de recursos físicos da máquina na qual o gerador de carga está, por exemplo, a aplicação informa que não pode operar, e solicita ao usuário que faça uma verificação de acordo com o motivo informado para a não operação.

É importante frisar que a troca de mensagens entre a ferramenta e o sistema TEF também acarreta o consumo de recursos físicos da máquina na qual o gerador está. Por esse motivo, é interessante realizar testes preventivos, a fim de determinar a quantidade de PDVs que podem ser acionados em uma só máquina antes de realizar experimentos, evitando assim, que o WGSysEFT interrompa sua operação devido à escassez de recursos físicos.

C.1 O WGServer

Os recursos físicos da máquina onde o WGSysEFT está localizado limitam a quantidade de PDVs que a ferramenta pode acionar com sucesso. Um meio de contornar este inconveniente é fazer com que a ferramenta possa atuar gerando carga a partir de diferentes máquinas. Para alcançar este objetivo o WGServer foi desenvolvido como uma ferramenta de apoio ao WGSysEFT.

O WGServer é uma aplicação que atua como servidor de apoio informando ao WGSysEFT números de identificação para cada PDV a ser ativado. Antes do WGSysEFT ser posto em atividade, é preciso acionar o WGServer.

Para acessar o WGServer, o WGSysEFT irá ler o arquivo AddressConfig.txt, que é disponibilizado junto com o WGServer para ser colocado na pasta onde esta instalado o WGSysEFT, obtendo o endereço do servidor de apoio, desta forma, cada máquina que contém o WGSysEFT deve conter o arquivo AddressConfig.txt devidamente configurado com o endereço IP da máquina onde o WGServer está executando.

O servidor de apoio pode coexistir na mesma máquina em que uma instância do WGSysEFT (Figura C.4 (1)). Dispensando assim, a necessidade da aplicação servidora habitar uma máquina exclusiva para tal tarefa, uma vez que consome poucos recursos computacionais. Neste caso, em específico, o arquivo AddressConfig.txt deve conter o

endereço IP da própria máquina.

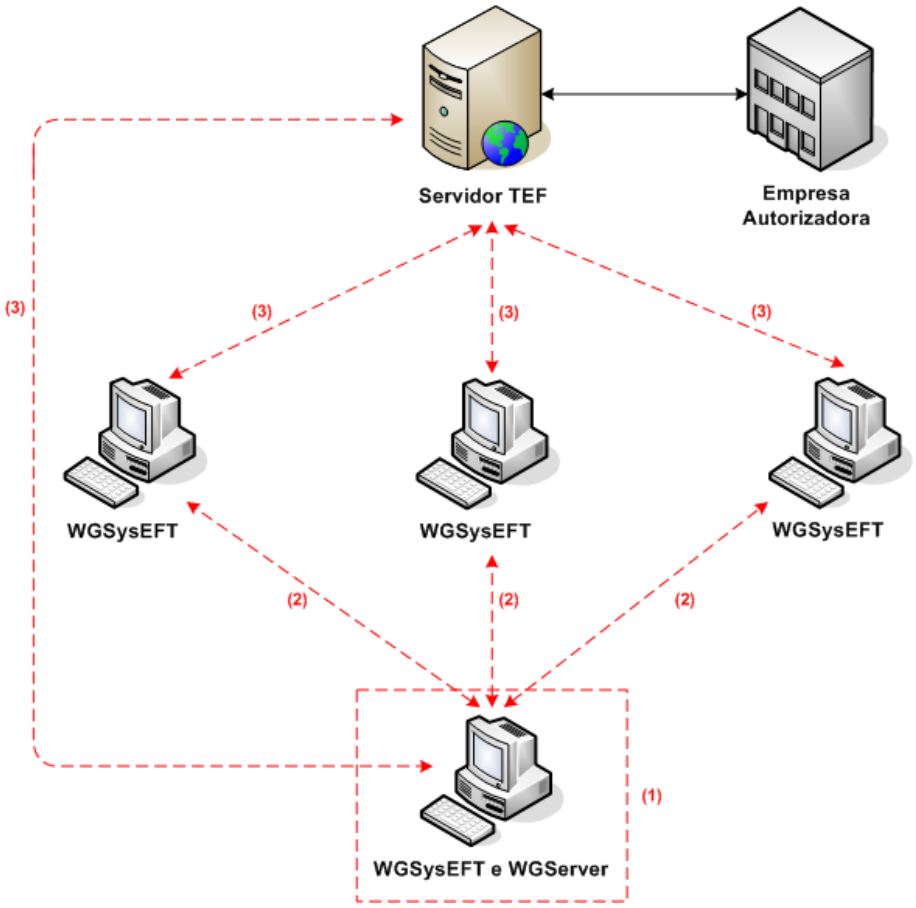


Figura C.4 Estratégia de injeção de carga envolvendo o WGServer

No instante em que o WGServer é ativado, este encontra-se apto a atribuir que PDV ficará com que numeração de identificação. O sistema TEF registra cada PDV ativado através de um número de identificação (números sequenciais de 1 até n a contar do primeiro PDV que foi ativado) e também através do endereço IP pertencente aos PDVs. Por exemplo, o PDV com o IP 192.16.0.16 é o terceiro PDV a ser ativado, portanto recebe o número de identificação 03. Quando o WGSysTEF entra em execução (fluxo 2, Figura C.4), faz-se uma busca para verificar se o WGServer está ativo, caso contrário o próprio WGSysEFT se encarrega de atribuir a numeração de identificação para cada PDVs que irá criar. Tal procura é feita por meio do endereço IP informado a cada instância da ferramenta antes de sua execução. Logo que o PDV é criado este envia uma mensagem contendo dados de registro para estabelecer comunicação com o servidor TEF. Embora o PDV possa fazer transações seguidas, a solicitação de comunicação só precisa ser enviada uma vez. Para cada transação a ser feita, uma sessão é aberta. Após

a conclusão, esta é encerrada, e outra sessão é aberta se a ferramenta (WGSysEFT) tiver em tempo hábil para realizar transações. Finalmente, na última etapa (fluxo 3, Figura C.4), o servidor TEF aceita as solicitações de transações, e dá prosseguimento ao fluxo de troca de mensagens para conclusão de transações.

A estratégia de injeção de carga concebida com o auxílio do WGServer propicia a geração de carga de forma descentralizada. Caso o fluxo de dados transmitido ao servidor TEF parta de apenas uma máquina, o uso do WGServer é dispensável.

D

Manual da Ferramenta EucaBomber

O EucaBomber trabalha com dados fornecidos por meio de sua *interface* gráfica. A Figura D.1, apresenta o *screenshot* da tela inicial da ferramenta, onde os dados que caracterizam os possíveis cenários a serem executados devem ser inseridos.

Como mostra a Figura D.1, os primeiros dados a serem informados (*Inform data*) correspondem ao endereço IP, MAC, usuário e senha, com permissão administrativa. O preenchimento desses campos é estritamente necessário, visto que o protocolo SSH2 necessita dessas informações para estabelecer a comunicação, exceto o MAC que é utilizado apenas para reparar as falhas ocasionadas. Mesmo que o usuário opte por injetar falhas sem fazer o reparo, o campo que corresponde ao endereço MAC também deve ser preenchido.

Ao menos um tipo de falha, presente no painel intitulado "*Type failure*", deve ser escolhido. Após a escolha, a ferramenta habilita o(s) campo(s) referente(s) ao(s) anteriormente selecionado(s), permitindo a inserção de dados. A seguir, o usuário deve definir se deseja que a ferramenta repare as falhas que ocasionar através das opções *Yes* ou *No* em "*Repair failure*". Se *Yes* for selecionado, então a ferramenta providencia os reparos. Caso contrário, fica a cargo do utilizador efetuar os reparos necessários.

Se falha de *software* for selecionado, então é necessário optar por um dos componentes de alto nível do Eucalyptus listados no combo box, são eles: *Cloud Controler*, *Cluster Controler*, *Node Controler*, *Storage Controler* e *Walrus*.

A seguir, é preciso escolher uma dentre as distribuições de probabilidade: Erlang, Exponencial, Geométrica, Log-normal, Normal, Pareto, Poisson, Triangular, Uniforme e Weibull. Cada distribuição possui um ou mais parâmetros que devem ser informados. A partir da escolha da distribuição, a ferramenta exibe os campos contendo o nome do parâmetro e o(s) campo(s) de texto a ser(em) inserido(s). Por exemplo, se a distribuição Triangular for escolhida, a ferramenta mostra os campos referentes aos parâmetros

mínimo (*minimum*), máximo (*maximum*) e moda (*mode*). Estes parâmetros devem ser preenchidos em formato de tempo. Isto é, o usuário determina se eventos devem ocorrer em meses, dias, horas, minutos, segundos ou milissegundos. A sintaxe adotada para informar o tempo limite (*time-out*) é: *M-x;D-x;H-x;m-x;s-x*, onde *M* corresponde ao mês, *D* ao dia, *H* a hora, *m* a minuto e *s* a segundo, enquanto *x* representa o valor para cada unidade de tempo mencionada.

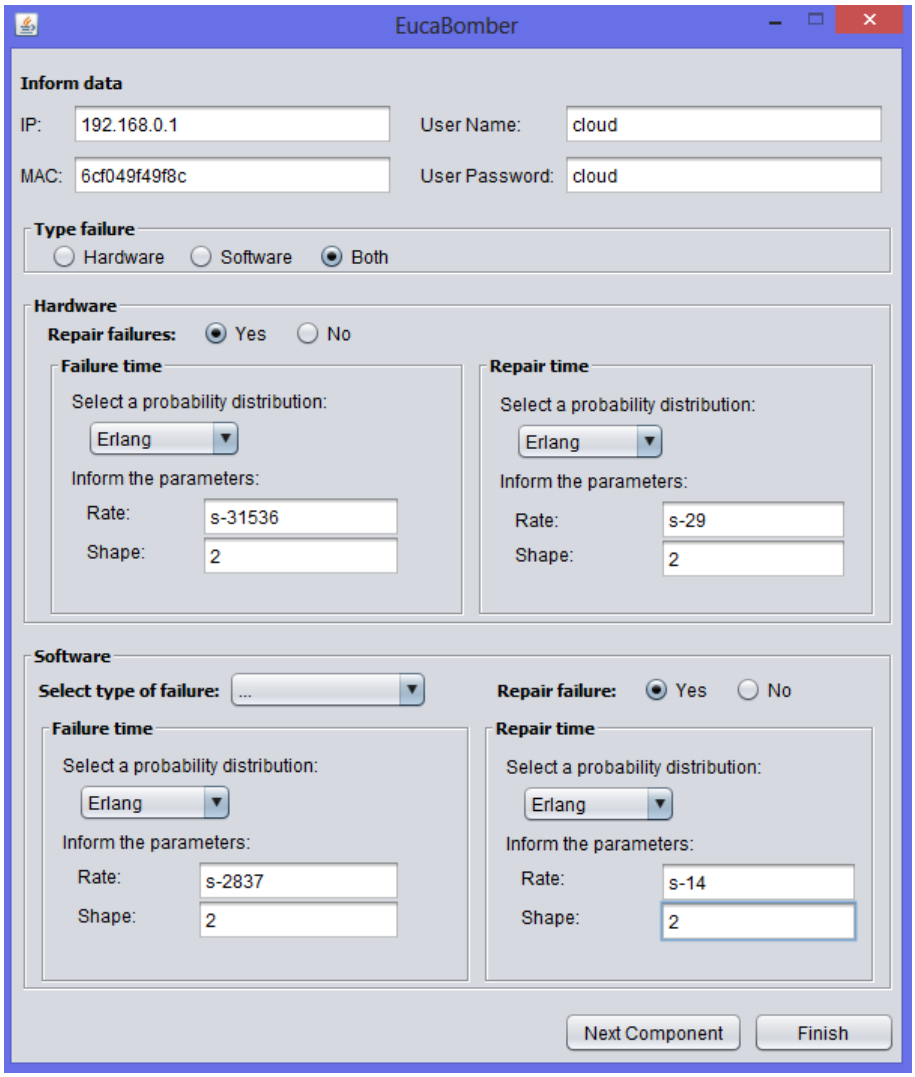


Figura D.1 *Screenshot* da primeira interface gráfica do EucaBomber

Algumas partes desta sintaxe podem ser suprimidas quando não necessárias, por exemplo, um usuário deseja definir eventos seguindo uma distribuição exponencial, com média de 2 horas entre os eventos. Neste caso, o tempo de espera pode ser definido com a sintaxe *H-2*. Em outro caso, se o evento ocorrer em média a cada 5 meses, 20 horas e

40 minutos, o tempo de espera pode ser expresso como *M-5;H-20;m-40*. É importante salientar que as unidades de tempo necessariamente devem ser separadas por ponto e vírgula (";"), sem espaços entre eles. A única exceção ao uso da sintaxe, ocorre quando o usuário deseja especificar o tempo limite em milissegundos. Neste caso, o usuário só precisa informar o valor numérico, sem qualquer outro símbolo. Por exemplo, um usuário pode informar o valor 1000 como o parâmetro taxa (*rate*) da distribuição exponencial, e o EucaBomber entende este valor como 1000ms. Parâmetros de distribuições específicas, que não têm qualquer unidade de tempo, também podem ser expressas com valores individuais, sem utilizar a sintaxe especificada. Um exemplo de tal caso ocorre quando um usuário define um evento por meio da distribuição Erlang, com uma taxa (*rate*) de 5 horas e 2 minutos e forma (*shape*) 2. A taxa é expressa como *H-5;m-2* considerando a forma simplesmente como *2*.

Após a escolha e preenchimento dos dados selecionados, na primeira *interface* gráfica do EucaBomber, o usuário é apresentado a duas alternativas para prosseguir com o uso da ferramenta: inserir novos dados para causar mais falhas, ou informar que já concluiu esta etapa de configuração.

O ferramental possibilita a inserção de novas falhas e o reparo destas, se desejado, acionando o botão "*Next Component*". Esta ação só poderá ser acionada quando a primeira tela for preenchida. A ação deste botão faz com que o EucaBomber guarde os dados inseridos na primeira tela e a apresente novamente ao usuário para que este possa inserir novas falhas, sejam elas destinadas a mesma máquina, ou para uma outra. A medida que falhas vão sendo informadas, a ferramenta apresenta um contador informando a quantidade incluída. Se, por ventura, a primeira inserção de dados de falhas estiver incompleta, a ferramenta informa qual campo deve ser preenchido para que uma nova opção de falha possa ser adicionada. Esta estratégia possibilita fornecer ao usuário a opção de informar vários tipos de falhas, se desejar, para diversas máquinas.

Caso o usuário não deseje incluir mais de dois tipos de falhas (*hardware* e *software*) para a mesma máquina, ou ainda que já tenha concluído a inserção das falhas desejadas, é preciso acionar o botão "*Finish*". Este botão salva o restante das informações adicionadas, e apresenta a segunda tela, onde o usuário pode verificar se todas as informações anteriormente preenchidas estão corretas. O *sreenshort* da segunda tela é apresentado na Figura D.2.

Como pode ser observado na Figura D.2, a parte superior da tela apresenta uma lista contendo todos os dados informados através da primeira *interface* gráfica da aplicação. Através desta lista, o usuário pode observar e realizar alterações caso algum dado tenha

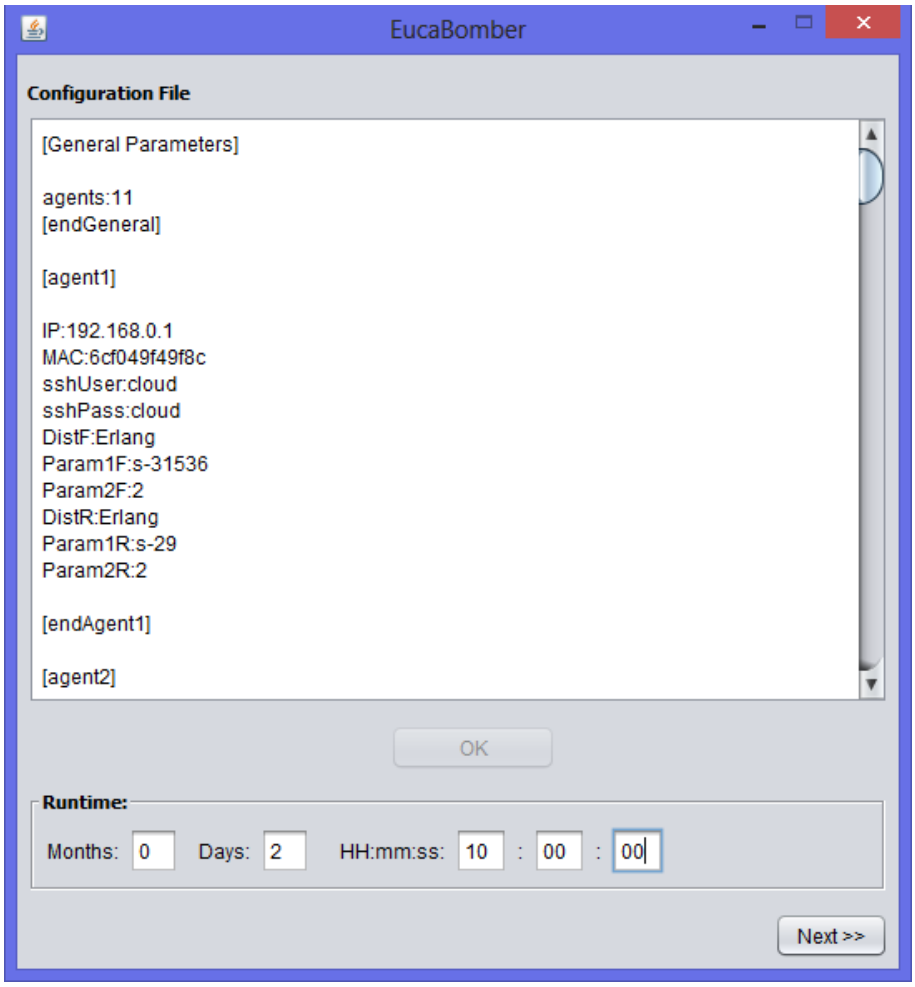


Figura D.2 *Screenshot* da segunda interface gráfica do EucaBomber

sido informado incorretamente. A lista possui a seguinte estrutura:

- **[General Parameters]:** cabeçalho que contém a quantidade de agentes que devem ser instanciados. Cada agente corresponde a uma falha e um reparo, este último se solicitado;
- **agents:** quantidade de agentes que devem ser criados;
- **[endGeneral]:** finaliza o cabeçalho *General Parameters*;
- **[agentX]:** inicia a delimitação por agente de falha e reparo, este último quando solicitado. Cada agente recebe um número após a palavra *agent*, em substituição ao *X*, como forma de identificação. Por exemplo: *agent1*, *agent2* e assim sucessivamente. No corpo desta instrução, encontram-se os dados informados pelo

usuário como o IP, MAC, usuário (sshUser), senha (sshPass), a distribuição escolhida (DistF – para falhas e DistR – para reparos) e os parâmetros passados (por exemplo, s-31536). Se a opção de reparo for selecionada, então neste espaço deve conter a distribuição escolhida, e os parâmetros configurados para estabelecer o reparo. Ainda nesta parte é estabelecido se o agente é responsável por ocasionar falhas de *hardware* ou *software*. Em caso de falha de *software*, o espaço citado conterá também uma linha com a palavra "*software*" escrita, e os comandos a serem executados. Caso contrário, não terá nenhum indicativo claro, ou seja, a falha será de *hardware*.

Após verificar se as informações estão corretas, o usuário deve acionar o botão "*ok*". A partir deste momento, o usuário fica impossibilitado de realizar qualquer tipo de modificação, e ocorre a liberação dos campos do painel "*runtime*". É a partir destes campos que o usuário informa o tempo que a ferramenta deve executar, estabelecendo meses, dias, horas, minutos e segundos. Ao menos um desses campos deve ser informado para dar início a execução da ferramenta, os demais são preenchidos com 0. Com a conclusão desta etapa, resta ao usuário selecionar o botão "*Next*" para acionar a terceira, e última, tela do EucaBomber. O *screenshort* da terceira *interface* gráfica da ferramenta pode ser vista na Figura D.3.

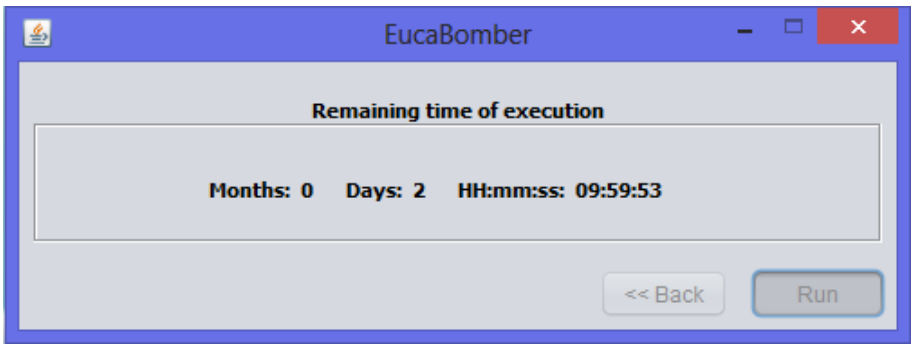


Figura D.3 *Screenshot* da terceira interface gráfica do EucaBomber

Na última tela é possível retornar a anterior ou iniciar a operação da ferramenta. Se o botão "*Back*" for selecionado, a aplicação volta para tela anterior. Contudo, a única alteração permitida é no tempo informado para operação da ferramenta. O botão "*Run*" aciona a ferramenta, e dispara um relógio em contagem regressiva para o termino de operação.

Ao final da execução, a ferramenta gera um arquivo no formato CSV (*comma-separated values*), reportando o comportamento de execução da aplicação ao longo do

experimento. Este arquivo, nomeado *Report*, contém dados como endereço IP das máquinas, o tipo de falha inserida, a ação de reparo (se for o caso) e os respectivos instantes de tempo em que os eventos ocorreram, sendo este último expresso em milissegundos. Um exemplo desse arquivo pode ser visto na Figura D.4.

1	Report Generator Failures and Repairs			
2				
3	Parameters selected generator failures and repairs			
4	Runtime: 0 months, 2 days and HH:mm:ss 00:00:00			
5	Configuration file selected: config.ini			
6				
7	Generator behavior during the execution:			
8	Agent number	IP number	instruction executed	when executed(ms)
9	Client Agent - 8	192.168.0.4	sudo service eucalyptus-nc stop	897741.91
10	Client Agent - 8	192.168.0.4	sudo service eucalyptus-nc start	28042.78
11	Client Agent - 10	192.168.0.5	sudo service eucalyptus-nc stop	1145664.76
12	Client Agent - 10	192.168.0.5	sudo service eucalyptus-nc start	11835.21
13	Client Agent - 11	192.168.0.1	sudo service eucalyptus-cc stop	1278750.39
14	Client Agent - 11	192.168.0.1	sudo service eucalyptus-cc start	39335.07
15	Client Agent - 4	192.168.0.2	sudo service eucalyptus-nc stop	1486197.83
16	Client Agent - 4	192.168.0.2	sudo service eucalyptus-nc start	14730.62
17	Client Agent - 2	192.168.0.1	sudo service eucalyptus stop	1974719.66
18	Client Agent - 6	192.168.0.3	sudo service eucalyptus-nc stop	1975553.67
19	Client Agent - 6	192.168.0.3	sudo service eucalyptus-nc start	2468.22
20	Client Agent - 2	192.168.0.1	sudo service eucalyptus start	40565.59
21	Client Agent - 10	192.168.0.5	sudo service eucalyptus-nc stop	2757753.32
22	Client Agent - 10	192.168.0.5	sudo service eucalyptus-nc start	2526.97

Figura D.4 Parte do arquivo Report gerado pelo EucaBomber