Pós-Graduação em Ciência da Computação

Aline Santana Oliveira

SIMF: UM FRAMEWORK DE INJEÇÃO E MONITORAMENTO DE FALHAS DE NUVENS COMPUTACIONAIS UTILIZANDO SPN

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

RECIFE

2017

Aline Santana Oliveira

SIMF: UM FRAMEWORK DE INJEÇÃO E MONITORAMENTO DE FALHAS DE NUVENS COMPUTACIONAIS UTILIZANDO SPN

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: Paulo Romero Martins Maciel

RECIFE
2017

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

O48s Oliveira, Aline Santana
SIMF: um *framework* de injeção e monitoramento de falhas de nuvens computacionais utilizando SPN / Aline Santana Oliveira. – 2017.
99 f.: il., fig., tab.

Orientador: Paulo Romero Martins Maciel.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2017.
Inclui referências e apêndices.

1. Ciência da computação. 2. Redes de Petri. 3. Computação em nuvem. I. Maciel, Paulo Romero Martins (orientador). II. Título.

004 CDD (23. ed.) UFPE- MEI 2017-253

Aline Santana Oliveira

SIMF: UM FRAMEWORK DE INJEÇÃO E MONITORAMENTO DE FALHAS DE NUVENS COMPUTACIONAIS UTILIZANDO SPN

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

Aprovado em: 25/08/2017.

BANCA EXAMINADORA

_____

Prof. Dr. Nelson Souto Rosa
Centro de Informática / UFPE

_____

Prof. Dr. Rosangela Maria de Melo
Departamento de Informática/IFPE

_____

Prof. Dr. Paulo Romero Martins Maciel
Centro de Informática / UFPE
(Orientador)

Dedico esta pesquisa a Deus, minha avó Belinha, e a minha família pelo apoio em todos os momentos.

Agradecimentos

Agradeço a Deus pela oportunidade que Ele tem me concedido, ao meu esposo pelo suporte que me ofereceu, pelos conselhos e por nunca me deixar desistir, aos meus pais Adão e Alaide pelos conselhos e inúmeras orações direcionadas a mim e pela confiança, ao meu irmão Athos pela amizade e apoio. Aos meus amigos do grupo MODCS em principal Jamilson pelo apoio e contribuição neste trabalho. A Rosângela, Renata, Camila, Iure, Jonas e Erico pela amizade sincera e troca de sorrisos constantes. A Raisa pela amizade construída. A minhas primas pelas mensagens em meio ao mestrado longe de "casa". E principalmente ao professor Dr. Paulo Maciel por ter me aceitado em seu grupo de pesquisa, pela orientação, os ensinamentos e a dedicação ao seu trabalho. Meu sincero obrigada!

"Não fui eu que lhe ordenei? Seja forte e corajoso! Não se apavore e nem desanime, pois o SENHOR, o seu Deus, estará com você por onde você andar."

(Josué 1:9)

Resumo

A computação em nuvem é um paradigma computacional que vem sendo utilizado ao longo dos últimos anos devido as suas características de provisionamento de recursos de forma escalável, onde seus usuários pagam apenas por aquilo que consomem. Esse modelo computacional possibilita que diversos serviços sejam ofertados a partir da sua Infraestrutura como Serviço. Porém, a falha de componentes dos recursos da nuvem é algo bastante comum e que afeta diretamente a disponibilidade dos serviços que os utilizam. Dessa forma, surgiu o interesse na área da pesquisa acadêmica em estudar e avaliar esse ambiente a fim de garantir alta disponibilidade em serviços na nuvem. Para auxiliar na avaliação desses serviços, os pesquisadores desenvolvem ferramentas, entretanto a maioria dos *softwares* precisam de atualizações constantes para que se adaptarem ao ambiente no qual foi desenvolvido o que leva o usuário ao retrabalho. Sendo assim, este trabalho tem como proposta desenvolver um *framework* que auxilie o pesquisador no estudo de disponibilidade de serviços de nuvem computacional. Esse *framework* utiliza SPN (*Stochastic Petri Nets*) como um mecanismo de injeção de falhas, que permite que o usuário avalie vários modelos de nuvens computacionais pois o *framework* não sofrerá modificação para se adequar ao ambiente computacional que será avaliado. Além disso a solução proposta monitora o ambiente e informa ao usuário os tempos de falha e reparo do sistema. Nossos resultados mostraram que o *framework* foi eficiente e eficaz no resultado da disponibilidade dos modelos avaliados no estudos de caso.

Palavras-chaves: Rede de Petri Estocástica. Injeção de Falha. Computação em nuvem. Monitoramento. Dependabilidade

Abstract

Cloud computing is a computational paradigm that has been used over the last few years because of its resource provisioning characteristics in a scalable way, where their users pay only for what they consume. This computational model enables several services to be offered from its Infrastructure as a Service. However, the failure of components of cloud resources is something quite common and that directly affects the availability of the services that use them. Thus, interest in the field of academic research has arisen in studying and evaluating this environment in order to guarantee high availability of services in the cloud. To assist in the evaluation of these services, researchers develop tools, however, most software requires constant updating to adapt to the environment in which the user is led to rework. Therefore, this work aims to develop a framework that helps the researcher in the study of the availability of cloud computing services. This framework uses SPN as a fault injection mechanism, which allows the user to evaluate several models of computational clouds because the framework will not be modified to suit the computational environment that will be evaluated. Moreover, the proposed solution monitors the environment and informs the user of the failure times and system repair. Our results showed that the framework was efficient and effective in the result of the availability of the models evaluated in the case studies.

Key-words: Stochastic Petri net. Fault Injection. Cloud Computing. Monitoring. Dependability

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

CC *Controlador de Cluster.*

CLC *Controlador de Nuvem.*

CTMC *Continuous Time Markov Chain.*

GSPN *Generalized Stochastic Petri Net.*

HW *Hardware.*

IaaS *Infraestrutura como Serviço.*

MTBF *Mean Time Between Failures.*

MTTA *Mean Time To Activate.*

MTTF *Mean Time to Failure.*

MTTR *Mean Time to Repair.*

NC *Controlador de Nó.*

NIST *National Institute of Standards and Technology.*

PaaS *Plataforma como Serviço.*

RBD *Reliability Block Diagram.*

SaaS *Software como Serviço.*

SC *Controlador de Armazenamento.*

SO *Sistema Operacional.*

SPN *Stochastic Petri Nets.*

Sumário

1 INTRODUÇÃO

Com a evolução dos sistemas computacionais sobrevém a necessidade de ferramentas que ofereçam aos seus usuários sistemas de fácil usabilidade com funcionalidades diversificadas, confiáveis, de baixo custo e de alto desempenho. O empenho empregado para que o sistema atenda estas características ocorre desde a fase inicial do processo de desenvolvimento até sua implantação. Assim é importante assegurar que os serviços providos sejam de sobremodo confiáveis e de alto desempenho (SILVA et al., 2013).

Em virtude disso, surgiu a computação em nuvem que nada mais é do que um conjunto de recursos virtualizados facilmente utilizáveis e acessíveis, tais como *hardware*, *software*, plataformas de desenvolvimento e serviços. Estes recursos podem ser dinamicamente reconfigurados para se adequarem à uma carga de trabalho variável, permitindo a otimização do seu uso (VAQUERO et al., 2008). Desta forma, manifesta-se uma preocupação com as infraestruturas de computação em nuvem relacionado ao seu funcionamento, visto que é de suma importância que a mesma funcione de forma ininterrupta.

A dependabilidade é uma propriedade que integra atributos como disponibilidade, confiabilidade, segurança, manutenabilidade, testabilidade e desempenho (MACIEL, Paulo R M; LINS, Rafael D;CUNHA, 1996). Quando utilizamos alguns destes atributos em estudos, estamos de certa forma contribuindo para que sistemas sejam justificadamente mais confiáveis. O estudo da dependabilidade de sistemas pode ser realizado por meio da ação de carga de trabalho (SOUZA, 2013), por conseguinte, o uso desta pode contribuir para revelar falhas antes não perceptíveis que possam existir em componentes do sistema. Outra maneira de efetuar estas analises é através da injeção de eventos de falhas, observando assim o comportamento do sistema mediante seus efeitos.

Baseado neste contexto, esta dissertação foi motivada pela necessidade de desenvolver uma ferramenta de geração de eventos de falha e reparo, que utilizasse como mecanismo para tais ações um modelo formal denominado SPN, provendo para a realização de experimentos nas áreas de avaliação de dependabilidade, especificamente na parte de disponibilidade, em plataformas de nuvens computacionais.

A partir disso, desenvolveu-se o *framework* SIMF (*SPN Injection and Monitoring of Failures Framework*), que emprega como mecanismo de falha e reparo, o formalismo SPN. Sua eficiência é observada por meio de cenários de *testbed*, onde foi possível analisar os dados obtidos e averiguar a disponibilidade de uma plataforma de nuvem privada. Os cenários de teste ilustram que a ferramenta pode colaborar com administradores e planejadores de nuvens computacionais a avaliar a disponibilidade do sistema e políticas de manutenção.

1.1 Motivação e Justificativa

Recentemente sobreveio um novo paradigma para distribuição de serviços de computação. Trata-se de um modelo semelhante à distribuição de serviços básicos como o fornecimento de eletricidade, água, telefone e assim por diante, onde o provisionamento desses serviços utiliza o modelo de pagamento baseado no uso, e toda sua infraestrutura entrega os serviços de forma transparente ao usuário. A esse novo paradigma convencionou-se chamar de Computação em Nuvem (CANEDO, 2013).

A computação em nuvem é um conjunto de recursos virtualizados facilmente utilizáveis e acessíveis, tais como *hardware*, *software*, plataformas de desenvolvimento e serviços. Estes recursos podem ser dinamicamente reconfigurados para se ajustarem à uma carga de trabalho variável, permitindo a otimização do seu uso. A implementação de uma nuvem privada no âmbito de uma empresa pode agregar diversos benefícios à mesma, tais como: i) melhorias no aproveitamento dos recursos, ii) redução dos custos com manutenção, iii) redução do consumo de energia, iv) permitir maior controle das configurações da nuvem; v) além de permitir que ela adquira experiência sobre o funcionamento de uma nuvem.

Em consequência, no âmbito da pesquisa encontram-se várias de ferramentas com a finalidade de avaliar nuvens computacionais fazendo uso de carga de trabalho (injeção de falhas). Contudo algumas precisam de modificações em seu código para adaptarem-se ao ambiente de estudo devido ao *software* ou infraestruturas sofrerem modificação com relação às suas versões. Tal fato faz com que equipes de desenvolvimento sejam incumbidas de criar ferramentas para medição e testes de dependabilidade constantemente.

Baseado neste contexto surgiu, à necessidade de desenvolver uma ferramenta de geração de eventos sintéticos que utilize um modelo formal categorizado de SPN, com o intuito de contribuir para a realização de experimentos na área de dependabilidade de sistemas. O ambiente SIMF utiliza rede de Petri Estocástica como um mecanismo de injeção de falha em uma infraestrutura de nuvem, além de monitorar o sistema, informando ao usuário a ocorrência de falhas e reparos e verificando a cada instante se a nuvem está funcional. Ao final do processo é gerado um relatório que serve como base para análise de disponibilidade do sistema em questão.

Esta ferramenta busca evitar que ocorra retrabalho por parte dos usuários na geração de ferramentas de injeção de falhas. Seu diferencial é a utilização de SPN para este intuito.

1.2 Objetivo

O objetivo principal desse trabalho é propor um ambiente que utilize rede de Petri Estocástica (SPN - *Stochastic Petri Net*) para injetar e monitorar falhas em nuvem computacional. Esta ferramenta busca diminuir o retrabalho por parte dos desenvolvedores que estudam métricas de disponibilidade, a fim de tornar-se uma ferramenta capaz de ser

adaptável a qualquer sistema de nuvem no qual for utilizada.

De forma resumida essa dissertação busca atender os seguintes objetivos específicos:

- Modelos estocásticos SPN utilizam fundamentação matemática, representação gráfica e possibilidade de simulações e verificações do sistema;
- Provê suporte adequado ao estudo de disponibilidade em plataformas de nuvens computacionais;
- Modelar estados de um sistema; e
- Validar modelos SPN de nuvens computacionais em um cenário real utilizando uma única ferramenta.

1.3 Trabalhos Relacionados

Podemos encontrar na literatura vários trabalhos envolvendo a criação e utilização de carga de trabalho sintética para avaliação dos mais variados tipos de sistemas. Entretanto nenhum dos referidos permeia todas as áreas presentes nesta dissertação, como injeção de falhas utilizando rede de Petri, monitoramento de uma nuvem privada e análise de modelos discretos. É importante salientar que alguns destes trabalhos podem se relacionar com uma ou mais áreas do conhecimento.

Levando em consideração a sustentabilidade e as necessidades atuais de energia, sem comprometer os recursos não renováveis para as gerações futuras, como por exemplo, a utilização de data centers os autores (SILVA et al., 2013) desenvolveram uma ferramenta denominada Astro que busca automatizar atividades de projetos, fornecendo resultados rápidos aos desenvolvedores. Seu ambiente contempla: Diagramas de bloco de confiabilidade (RBD), rede de Petri Estocástica (SPN), cadeias de Markov contínuas (CTMC) para avaliação de depentabilidade e um método baseado na avaliação do ciclo de vida (ACV) para a quantificação do impacto da sustentabilidade. ASTRO foi desenvolvido para avaliar infraestruturas de data center seu ambiente é genérico o suficiente para testar outros sistemas.

(SOUZA, 2013) trata-se de um *framework* chamado de *Flexloadgenerator* que deu origem a duas ferramentas. A primeira que analisa sistemas TEF na área de desempenho e a segunda denominada de *EucaBomber* atuando em plataforma de computação de nuvem *Eucalyptus*. O *framework* simula um estado de interrupção da execução do sistema, de modo que essas interrupções possam ser reparadas. As perturbações causadas pela inserção de falhas no sistema de nuvem afetam 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.

(SILVA et al., 2015b) desenvolveu uma ferramenta que oferece apoio para os estudos de depentabilidade e desempenho de sistemas em geral. A ferramenta denominada : [1] *Mercury*, permite a criação e avaliação de modelos: cadeias de Markov (CTMC), diagrama de blocos de confiabilidade (RBD), modelos de fluxo de Energia (EFM) e *Stochastic Petri Nets* (SPN).

(OLIVEIRA, 2014) propõe uma metodologia para a avaliação de disponibilidade e consumo energético em ambientes de *mobile cloud*.

Após validação do modelo para a arquitetura básica, (OLIVEIRA, 2014) propôs extensões desta arquitetura, sendo elas: i) *store and forward*, ii) múltiplas interfaces de rede, iii) e arquitetura *cloudlet*. Além disso, o mesmo também investigou o impacto de mecanismos de redundância na infraestrutura de nuvem sobre a disponibilidade e o *downtime* anual.

Para avaliar o impacto da conectividade sem fio sobre o consumo energético e disponibilidade, (OLIVEIRA, 2014), criou-se um modelo de disponibilidade na *cloudlet* em rede de Petri estocástica (SPN), e a partir deste modelo, foi gerado outro modelo focado em consumo energético, que possibilitava ao autor avaliar o tempo de vida da bateria, considerando o uso das redes Wi-Fi e 3G (simultaneamente). Os autores utilizaram mecanismos de redundância através de modelo rede de Petri (SPN) nos *Nós* do *cluster* para implantação de VMs (*virtual machine*).

O resultado obtido a partir deste modelo indicou uma melhora significativa na disponibilidade para os cenários que possuem múltiplas interfaces de rede. Entretanto, quando foi considerado o uso deste mecanismo em conjunto com a arquitetura baseada em *cloudlet*, a melhora foi insignificante, além disso, não foi realizado o mecanismo de redundância através do experimento com injeção de falhas e também não foi efetuado estudo de desempenho e performabilidade nas arquiteturas de *mobile cloud*.

A ferramenta criada por (GALINDO et al., 2009), chamada de WGCap (*Workload Generator for Capacity Advisor*), consiste em um gerador de carga de trabalho sintética, que é voltado para o planejamento de capacidade em sistemas de servidores virtuais VSE (*Virtual Server Environment*), essa solução foi pertencente a empresa HP (*Hewlett-Packard*), e 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, pois de acordo com (GALINDO et al., 2009) 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*, ou seja em carga de trabalho.

Para (BARAZA et al., 2000) o problema de analisar o comportamento de sistemas to-

[1] Ferramenta para estudo de formalismos matemáticos

lerantes a falha, ocorre pelo fato dos mesmos decorrerem de validação devido as constantes inovações dos componentes do sistema, tanto *hardware* como *software*. Ainda segundo (BARAZA et al., 2000), os sistemas tolerantes a falha são normalmente utilizados em aplicações especificas. Por sua vez, essas constantes mudanças dificultam a disponibilidade de dados experimentais relacionados a sua função. Torna-se então difícil quantificar a influência de falhas no sistema e assim avaliar a confiabilidade do mesmo. Pensando nisso, os autores desenvolveram um protótipo automático de uma ferramenta de injeção de falhas, para ser utilizado em uma plataforma da IBM-PC ou outra compatível.

A ferramenta foi construída através de um simulador VHDL, foi implementada com o intuito de gerar diferentes técnicas de injeção de falhas. Outro aspecto observado na ferramenta é o fato ser possível fazer uma análise dos resultados obtido a partir da injeção de falha, para compreender o modelo do sistema e/ou validar seu mecanismo de tolerância à falha. Alguns resultados obtidos dos testes de injeção de falhas foram realizados para validar a confiabilidade do sistema de um microcomputador. Dessa forma, foi analisado a patologia dos erros encontrados, medição da latência, e foi calculada também a quantidade de erros e latências de recuperação. No entanto a ferramenta possui algumas divergências, tais como: i) o tempo de simulação é demorado, ii) com relação ao contexto de falha de modelagem, seria interessante que fosse aplicado injeção de falhas a nível rede sem fio Wi-Fi em *software* e *hardware*, baseado em VHDL.

(FUJITA et al., 2012) propôs um conjunto de ferramentas para avaliação de dependabilidade, chamado de *DSBench Toolset*. O *DS-Bench* é composto por: *DCase Editor*, *DS-Bench* e *D-Cloud*, onde, juntos, estes módulos promovem a obtenção de métricas de dependabilidade do sistema fazendo uso de *benchmarks*, no qual estes *benchmarks* executam testes no sistema por completo, tanto em sistema operacional, rede, entre outros.Os autores afirmam que os testes podem ser executados em ambiente contendo tanto máquinas físicas como virtuais.

(COSTA, 2015) realizou um estudo de avaliação da disponibilidade de ambientes MBaaS (*Mobile Backend-as-a-Service*), utilizando a plataforma de MBaaS *open source* *OpenMobster*. O autor investigou cinco diferentes cenários, de modo que fosse possível verificar a viabilidade de cada um em relação à disponibilidade. Para validação desse sistemas, foram utilizadas técnicas de injeção de falhas em sistemas físicos a nível de *software*, com gatilho baseado em *time-out*. O objetivo foi verificar o comportamento da plataforma de MBaaS *OpenMobster* em um ambiente sujeito a falhas. Para a caracterização e análise do ambiente foi utilizado apenas componentes que são integrantes da plataforma de MBaaS *OpenMobster* e que estariam em execução: o Banco de Dados, a JVM, o JBoss e a própria plataforma. Os resultados obtidos (COSTA, 2015), podem auxiliar os administradores de ambientes MBaaS na utilização de uma infraestrutura de nuvem semelhante a sua proposta. Por fim, este trabalho apresenta uma análise detalhada da disponibilidade de diversas arquiteturas baseadas em nuvem privada.

Em (SILVA, 2016) propôs um *framework* denominado GeoClouds Modcs avaliar sistemas de migração em nuvem implantados em *data centers* geograficamente distribuídos, relativos à máquinas virtuais (VM) com foco em ocorrência de desastres em diferentes cargas de usuários. Além disso, fez uso de métricas de desempenho no qual levam em consideração as perspectivas do usuário e do provedor. Um conjunto de modelos estocásticos é utilizado para apoiar a avaliação, os modelos são divididos em blocos que podem ser combinados para criar modelos maiores.

Além disso, um método de validação é proposto para o sistema de computação em nuvem que considera as operações e a dependência do ciclo de vida da VM entre os componentes da nuvem. Os resultados do modelo são validados por adotando o *Eucabomber 2.0* em um ambiente real.

A ferramenta D-Cloud apresentada (BANZAI et al., 2010) é um sistema de nuvem que gerencia máquinas virtuais com instalação de injeção de falhas. A D-Cloud configura um ambiente de teste nos recursos da nuvem usando um determinado arquivo de configuração do sistema e executa vários testes automaticamente de acordo com um determinado cenário. Nesse cenário, o D-Cloud permite o teste de tolerância a falhas, causando falhas do dispositivo pela máquina virtual. O referido utilizou o *software Eucalyptus* e uma descrição para a configuração do sistema e o cenário de injeção de falhas escrito em XML. Descobriu-se que o sistema D-Cloud permite que o usuário configure e teste facilmente um sistema distribuído na nuvem e efetivamente reduz o custo e o tempo de teste.

O objetivo proposto nesta dissertação é desenvolver um *framework* que utilize rede de Petri Estocástica (SPN - *Stochastic Petri Net*) para injetar e monitorar falhas em nuvem computacional. Este *framework* busca diminuir o retrabalho por parte dos desenvolvedores que estudam métricas de disponibilidade, a fim de tornar-se uma ferramenta capaz de ser adaptável a qualquer sistema de nuvem no qual for utilizada.

Até o presente momento, não encontrou-se trabalhos que utilizassem a abordagem empregada nesta pesquisa. Um dos trabalhos que aproxima-se deste estudo é a ferramenta EucaBomber, entretanto a mesmo foi desenvolvida para infraestrutura de nuvem *Eucalyptus* e trata-se somente de um injetor de falhas e não utiliza modelos formais. Enquanto esta pesquisa utiliza um modelo SPN como um mecanismo de injeção de falhas e pode ser utilizado em várias infraestrutura de nuvem sem modificações no código.

A Tabela1 resume as principais contribuições desta pesquisa e em que ela se diferencia dos demais trabalhos apresentados até aqui.

Tabela 1 – Principais contribuições desta Dissertação

	Contexto	Injeção de Falha /Monitoramento	Métricas de Dependabilidade	Modelos Formais	Validação
(SILVA et al., 2015a)	Sistemas em Geral		✓	✓	
(SILVA et al., 2013)	*Data Center*		✓	✓	
(SOUZA, 2013)	Computação em Nuvem	✓	✓		
(OLIVEIRA, 2014)	*Mobile Cloud Computer*	✓		✓	✓
(GALINDO et al., 2009)	Servidores Virtuais	✓			
(BARAZA et al., 2000)	Plataformas IBM	✓			
(FUJITA et al., 2012)	Sistemas de Tempo Real	✓	✓		
(COSTA, 2015)	*Mobile Cloud Computer*	✓	✓		✓
(SILVA, 2016)	*Data Center* e Computação em Nuvem	✓	✓	✓	
(BANZAI et al., 2010)	Computação em Nuvem		✓		
Esta dissertação	SPN e Computação em Nuvem	✓	✓	✓	✓

1.4 Estrutura da Dissertação

Este documento está dividido em sete capítulos. Após este Capítulo, o Capítulo 2 apresenta os conceitos básicos no qual este trabalho está envolvido, tais como injeção de falha, dependabilidade e modelos matemáticos. O Capítulo 3 apresenta em detalhes os tipo de modelos existentes, para análise de sistemas. O Capítulo 4 abrange metodologia proposta neste trabalho. O Capítulo 5 trata do *framework* SIMF, seu funcionamento e os diagramas de classe e de sequência utilizados para seu desenvolvimento. O Capítulo 6 apresenta os estudos de caso realizados. O Capítulo 7 apresenta as considerações finais, listando as contribuições, e apresentando direcionamentos futuros.

2 FUNDAMENTOS

Este Capítulo apresenta uma visão geral dos conceitos com maior relevância utilizados neste trabalho. É válido ressaltar, que serão fornecidos os subsídios para o entendimento dos conceitos aplicados e que cada tema poderá ser melhor explorado, a partir das referências mencionadas. Inicialmente, são abordados os conceitos básicos sobre Computação em Nuvem, posteriormente *Frameworks* orientados a objetos. Em seguida, são apresentadas informações sobre Dependabilidade. Logo após, Modelos de avaliação com foco em rede de Petri e Injeção de falhas.

2.1 Computação em Nuvem

Várias definições têm sido propostas pelo meio acadêmico e pela indústria para computação em nuvem. Segundo o *National Institute of Standards and Technology* (NIST), a computação em nuvem é um modelo que permiti o acesso, de modo conveniente e sob demanda, a um *pool* compartilhado de recursos computacionais configuráveis (por exemplo, redes, servidores, armazenamento, aplicativos e serviços) que podem ser rapidamente provisionados e liberados com mínimo esforço de gerenciamento ou interação com o provedor de serviços (MELL; GRANCE et al., 2011). Segundo (VAQUERO et al., 2008), estes recursos podem ser dinamicamente reconfigurados para se ajustarem a uma carga de trabalho variável, permitindo a otimização do seu uso.

Os serviços disponibilizados na nuvem são: *softwares*, plataformas e infraestrutura. Os usuários, por sua vez, precisam apenas de um dispositivo conectado a internet para ter acesso a esses serviços ou por vezes, pagar pelo que consome. Ainda segundo (VELTE et al., 2010) a computação em nuvem vem despertando interesse em grandes empresas que passaram a investir em pesquisas sobre o tema na tentativa de conquistar espaço e tornar-se referência, podemos citar a *Google*, *IBM*, *Amazon*, *Microsoft* e *Apple*.

A computação em nuvem foi desenvolvida com o objetivo de oferecer serviços de fácil acesso, baixo custo e garantir características tais como disponibilidade e escalabilidade. Para (VELTE et al., 2010) esse modelo visa fornecer basicamente três benefícios: redução de custo na aquisição e composição da infraestrutura para atender as necessidades das empresas; Oferecer flexibilidade na adição e troca de recursos computacionais, permitindo escalar tanto em nível de recurso de *hardware* como de *software* para atender as demandas dos consumidores; E por fim facilitar o acesso aos serviços, uma vez que os usuários não precisam saber dos aspectos de localização física e de entrega dos resultados destes serviços.

Além das características essenciais que uma nuvem deve possuir, a mesma também é composta por três modelos de serviços (MELL; GRANCE et al., 2011). Que podem ser

conferidos na Figura 2.

- ***Software como Serviço*** **(SaaS)**: Refere-se a aplicações disponibilizadas pelo provedor como serviços e acessados pelos usuários através de um *browser*. O público alvo deste serviço são usuários finais, que poderão ter acesso aos softwares sem se preocuparem com a aquisição de licenças.

- ***Plataforma como Serviço*** **(PaaS)**: É o provisionamento de toda uma plataforma para desenvolvimento de aplicativos, abstraindo dos desenvolvedores os requisitos de hardware e possivelmente de outras camadas de software necessárias como banco de dados, servidor web e o suporte a linguagem de programação agilizando e reduzindo a complexidade do desenvolvimento. Como exemplos desse tipo de serviços pode-se citar a *AppEngine* do *Google* e o *Windows Azure* da Microsoft, onde o público alvo deste serviço são desenvolvedores de aplicações.

- ***Infraestrutura como Serviço*** **(IaaS)**: É responsável por prover toda a infraestrutura para a PaaS e o SaaS, onde seu principal objetivo é viabilizar o fornecimento de recursos, tais como servidores, rede, armazenamento e outros recursos de computação fundamentais para construir um ambiente de aplicação sob demanda, que podem incluir sistemas operacionais e aplicativos.

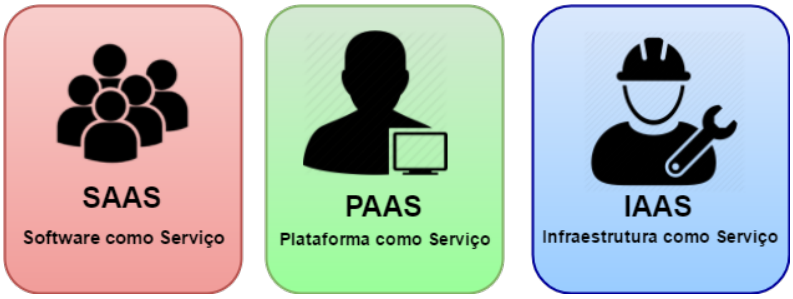

Figura 1 – Modelo de Serviço

A categorização da computação em nuvem, segue conforme o modelo de implantação, e leva-se em consideração o público-alvo e a abrangência da nuvem, que podem ser classificadas conforme a Figura 2. Além isso, existe outra categoria de nuvem que não é comumente citada, a nuvem comunitária é baseada no compartilhamento por diversas organizações (universidades ou indústrias), utilizada por uma comunidade específica que possui interesses semelhantes no que diz respeito à segurança, tais como missão e área de atuação (MELL; GRANCE et al., 2011):

- **Pública**: É a infraestrutura disponibilizada para o público em geral, podendo ser acessada por qualquer usuário. Neste caso a nuvem está hospedada no *datacenter* do provedor e este disponibiliza os serviços aos usuários de modo geral.

- **Privada**: É uma infraestrutura de nuvem usada exclusivamente por uma organização, sendo uma nuvem local ou remota gerenciada pela própria empresa ou por terceiros, onde a nuvem é operada exclusivamente pelo cliente e todos os seus dados estão localizados no seu próprio *datacenter*.

- **Híbrida**: É uma composição de duas ou mais nuvens (privada, comunitária ou pública) que permanecem como entidades únicas, ligadas por uma tecnologia padronizada ou proprietária que permite a portabilidade de dados e aplicações. Os diferentes tipos de nuvem envolvidos neste modelo podem estar hospedados tanto na infraestrutura do provedor quanto do cliente.

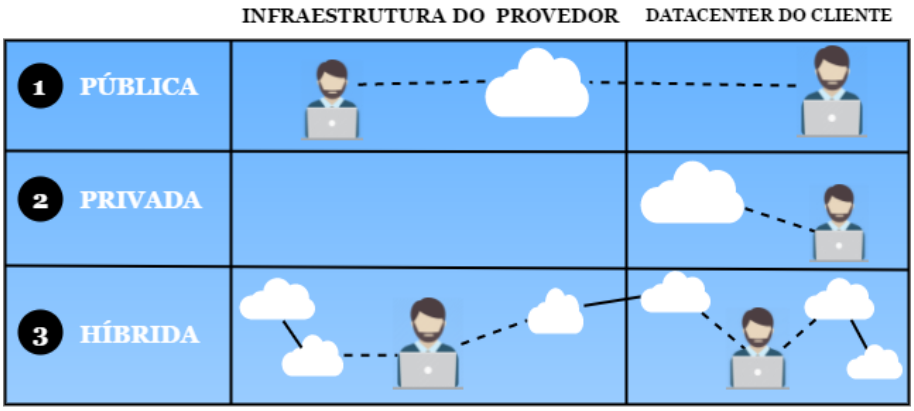

Figura 2 – Modelo de Implementação de Nuvem

2.2 Framework Orientado a Objeto

Na literatura existem vários trabalhos que definem o que é um *framework*. De acordo com (JOHNSON; FOOTE, 1988) *framework* é um conjunto de objetos que colaboram para realizar um conjunto de responsabilidades para um domínio de aplicação ou subsistema. Para (FAYAD; SCHMIDT; JOHNSON, 1999) *framework* é definido como um conjunto de classes que absorve um projeto abstrato de soluções para uma família de problemas relacionados. Ainda segundo (JOHNSON; FOOTE, 1988) o *framework* é um *design* abstrato para um tipo particular de aplicação, e geralmente consiste em um número de classes. Essas classes podem ser retirados de uma biblioteca ou podem ser específicos de aplicação.

Para (SOUZA, 2013) *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. Diferente de uma biblioteca comum em um ambiente de programação, no qual pode agregar um conjunto de classes que podem ser utilizadas separadamente, o *framework* é composto por classes que interagem umas com as outras. Uma diferença significativa entre bibliotecas e *framework*, encontra-se

na forma de construção, enquanto bibliotecas são implementadas principalmente focando em reuso.

No processo de desenvolvimento do *framework*, deve-se produzir uma estrutura de classes com a capacidade de adaptar-se a um conjunto de aplicações diferentes. A principal característica buscada ao desenvolver um *framework* é a generalidade em relação aos conceitos e funcionalidades do domínio tratado. Além disso, é fundamental que a estrutura produzida seja flexível. Pode-se afirmar, que o desenvolvimento de um *framework* é diferente do desenvolvimento de uma aplicação padrão. A distinção mais importante é que *framework* precisa cobrir todos os conceitos relevantes do domínio, enquanto uma aplicação se preocupa somente com os conceitos mencionados nos requisitos da aplicação (SOUZA, 2013).

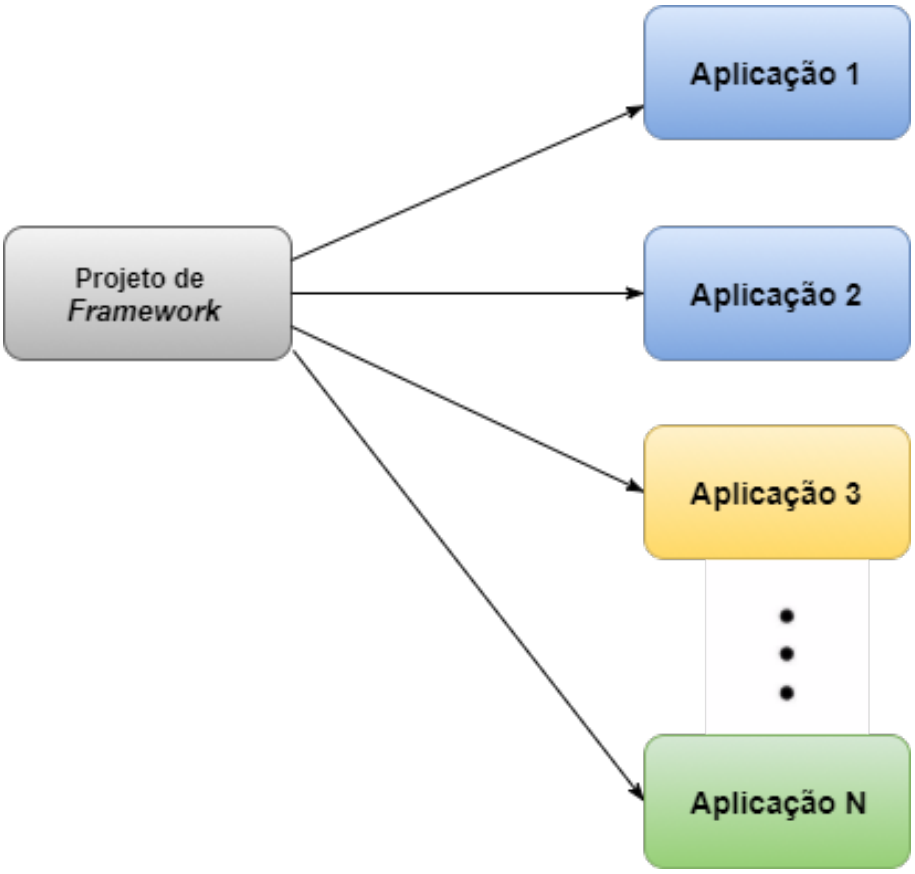

Figura 3 – Desenvolvimento de um *framework* - Adaptado (MATTSON, 1996)

2.2.1 Caracterização e Desenvolvimento de *Frameworks*

Os *frameworks* orientados a objetos podem ser classificado por diferentes domínios. Para (MATTSON, 1996) cada domínio aborda um problema especifico e podemos caracterizá-los como:

- *Framework* de suporte: Oferecem serviços de sistema de baixo nível, como dispositivos de drives de dispositivo e acesso a arquivos. Os desenvolvedores de aplicações utilizam *frameworks* de suporte diretamente ou usam modificações produzidas pelos provedores de sistema.

- *Framework* de aplicações: São estruturas que encapsulam a funcionalidade que pode ser aplicada a diferentes programas. Exemplos interface de usuário gráfica para aplicações comerciais.

- *Framework* de domínio: Encapsulam o conhecimento e experiência em um domínio de problema particular. Estruturas para controle de produção e multimídia são exemplos de *frameworks* de domínio.

O *Framework* utiliza-se de uma representação física no que se refere a termos de classes e métodos, onde não apenas a implementação é reutilizável, mas também a sua estrutura é reutilizável. A estrutura interna do mesmo está relacionada aos conceitos de arquiteturas de *software*. (MATTSON, 1996) descreve as arquiteturas de *framework* como:

- *Framework* arquitetural em camadas: Estrutura aplicações que possuem a capacidade de serem decomposta em grupos de sub-tarefas em diferentes níveis de abstração;

- *Framework* arquitetural *Pipes e filters*: É um padrão conhecido para organizar e executar componentes com dependências sequenciais é usado para executar várias tarefas de grandes fluxos de dados;

- *Framework* arquitetural MVC (*Model-View-Controller*): Definição de uma arquitetura para aplicações interativas que separa a interface do usuário de seu núcleo funcional;

- *Framework* arquitetural PAC (*Presentation-Abstraction-Controller*): Definição de uma estrutura para sistemas na forma de uma hierarquia de agentes cooperativos;

- *Framework* arquitetural reflexivo: Utilizado para aplicações que precisam considerar uma futura adaptação às mudanças de ambientes, tecnologia e exigências, porém sem uma modificação explícita de sua estrutura e de implementação;

- *Framework* arquitetural *microkernel*: Utilizado para sistemas que oferecem diferentes métodos sobre suas funcionalidades e que em alguns momentos precisam ser adaptados às novas exigências de sistemas;

- *Framework* arquitetural *blackboard*: Utilizado para estruturar aplicações complexas que envolvem vários subsistemas utilizado para diferentes domínios;

- *Framework* arquitetural *broker*: Utilizado aplicações distribuídas, onde uma aplicação pode acessar serviços de outras aplicações simplesmente pelo envio de mensagens a objetos mediadores, sem se preocupar com questões específicas relacionadas à comunicação entre processos, como a sua localização.

Os *frameworks* não podem ser vistos como bibliotecas de classes, visto que os mesmos são projeto de um conjunto de classes, que colaboram para realizar um conjunto de atividades. Enquanto os componentes das bibliotecas de classes são utilizados individualmente, as classes no *framework* são reutilizadas como um todo para resolver uma instância específica de um certo problema (LAJOIE; KELLER, 1995).

Para desenvolver-se o referido, é necessário produzir uma estrutura de classes com a capacidade de adequar um conjunto de aplicações diferentes. Uma das principais características que são utilizadas ao desenvolvê-lo é a generalidade em relação aos conceitos e funcionalidades do domínio tratado. É fundamental que a estrutura produzida seja flexível sendo assim, podemos afirmar que o desenvolvimento de um *framework* possui uma particularidade com relação ao desenvolvimento de uma aplicação padrão. Por outro lado para (FAYAD; SCHMIDT; JOHNSON, 1999) o diferencial mais importante é que *framework* precisa cobrir todos os conceitos relevantes do domínio enquanto uma aplicação, ou seja, o mesmo preocupa-se somente com os conceitos mencionados nos requisitos da aplicação.

A segunda maneira de desenvolvimento deste 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 maneira 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* (SOUZA, 2013). Enquanto na segunda maneira, vícios de projeto de aplicações anteriormente desenvolvidas serão repassados as próximas aplicações instanciadas a partir do mesmo.

Os *frameworks* tradicionais baseiam-se num quadro único, que abrange parte de aplicações específica. Nos últimos anos o desenvolvimento de aplicações está restringindo-se cada vez mais, muitas são baseadas em múltiplas bibliotecas e classes existentes. Este processo de composição pode levar a problemas de integração, uma vez que os *frameworks* são projetados com base na perspectiva tradicional, este é projetado para extensão (não composição) e sem legado para os componentes que devem ser incorporados.

2.3 Dependabilidade

Em (AVIZIENIS; LAPRIE; RANDELL, 2001) o termo dependabilidade é definido como a habilidade de um sistema computacional de entregar ou fornecer um serviço de maneira justa e confiável. Esta definição de alto nível considera a dependabilidade como uma

grande caixa preta que só pode ser analisada do ponto de vista do usuário do sistema, e sua avaliação será dada com base no comportamento esperado desta caixa, e das respostas por ela fornecidas a cada entrada realizada.

Podemos observar uma exposição sistemática dos conceitos da depentabilidade, que consiste em três aspectos: **atributos**, **meio** e **ameaças**, Figura 4 (AVIZIENIS; LAPRIE; RANDELL, 2001).

- **Atributos**: Possibilitam a obtenção de medidas quantitativas, que muitas vezes são cruciais para a análise dos serviços oferecidos.
- **Meios**: São os meios pelos quais a dependabilidade é atingida.
- **Ameaças**: Compreendem as falhas, erros e defeitos. A falha do sistema representa o evento que ocorre quando a entrega do serviço não acontece da maneira desejada.

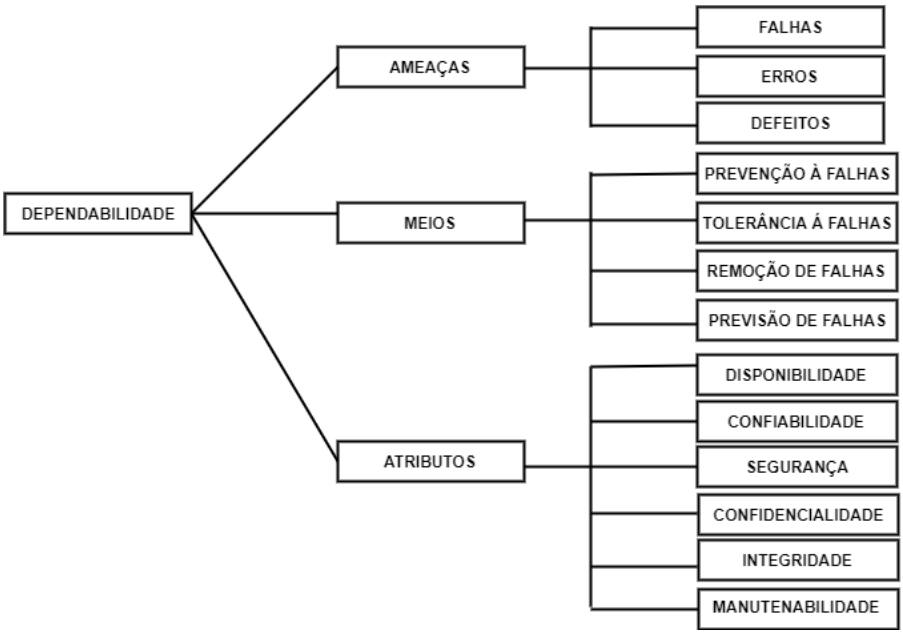

Figura 4 – Árvore de Dependabilidade - Adaptado (AVIZIENIS; LAPRIE; RANDELL, 2001)

A dependabilidade está diretamente ligada ao estudo do efeito de erros, faltas e falhas no sistema, a ocorrência destes ocasionam um impacto negativo nos atributos de dependabilidade. Técnicas de prevenção, predição, remoção e tolerância a faltas contribuem para manter níveis desejados de dependabilidade, sobretudo em sistemas críticos (AVIZIENIS; LAPRIE; RANDELL, 2001). De acordo com (WEBER, 2003), podemos caracteriza-los como:

- **Prevenção de falhas**: Impede a ocorrência ou introdução de falhas. Envolve a seleção de metodologias de projeto e de tecnologias adequadas para os seus componentes;

- **Tolerância a falhas**: Fornece o serviço esperado mesmo na presença de falhas. Técnicas comuns: mascaramento de falhas, detecção de falhas, localização, confinamento, recuperação, reconfiguração e tratamento;
- **Validação**: Remoção de falhas e verificação da presença de falhas;
- **Previsão de falhas**: Estimativas sobre presença de falhas e estimativas sobre consequências de falhas.

Os atributos de dependabilidade estão ligados as metas a serem alcançadas por sistemas que almejam ser considerados confiável. Um destes atributos, a disponibilidade, recebe um destaque maior em sua explanação, por tratar-se do foco desta dissertação de mestrado.

A disponibilidade é a probabilidade que o sistema esteja operacional durante um determinado período de tempo (MACIEL et al., 2012). Esta probabilidade pode ser dada através da razão entre o tempo de funcionamento esperado do sistema e a soma entre o tempo de funcionamento e o tempo de falha esperado (MACIEL et al., 2011).

Para (MACIEL et al., 2011) disponibilidade estacionária pode ser definida como: A razão entre o tempo de funcionamento esperado e a soma dos tempos de funcionamento e falha esperados. E pode ser observada segundo a formula abaixo:

$$A = \frac{E[Uptime]}{E[Uptime] + E[Downtime]} \tag{2.1}$$

, onde:

- **A** é a disponibilidade estacionária do sistema;
- **E[Uptime]** tempo em que o sistema tende a ficar disponível;
- **E[Downtime]** tempo em que o sistema tende a ficar indisponível;

De maneira contrária a indisponibilidade de um sistema é calculada pelas equações:2.2 e 2.3, na equação 2.3 **UA** significa a indisponibilidade do sistema enquanto o **A** é a disponibilidade (em inglês *Availability*).

$$UA = \frac{E[Downtime]}{E[Uptime] + E[Downtime]} \tag{2.2}$$

$$UA = 1 - A \tag{2.3}$$

Com base no valor da indisponibilidade é possível obter o downtime anual do sistema. O *downtime* anual representa o número esperado de horas que o sistema estará indisponível no intervalo de um ano, é calculado pela fórmula da Equação abaixo:

$$D = UA \times 8760h \tag{2.4}$$

A disponibilidade é frequentemente representada em **número de noves**. O número de noves corresponde a contagem dos algarismos consecutivos iguais a 9, após a vírgula (MARWAH et al., 2010).

$$N = \log_{10}(1 - A) \tag{2.5}$$

Quanto maior o número de noves, menor a indisponibilidade e *downtime* anual do mesmo, e provavelmente, mais complexos e custosos implementar mecanismos de redundância para manter a alta disponibilidade. Quando os tempos de *uptime* e *downtime* não estão disponíveis, geralmente são usados os valores médios entre os eventos de falha e reparo do sistema. Como podemos observar na Figura 5.

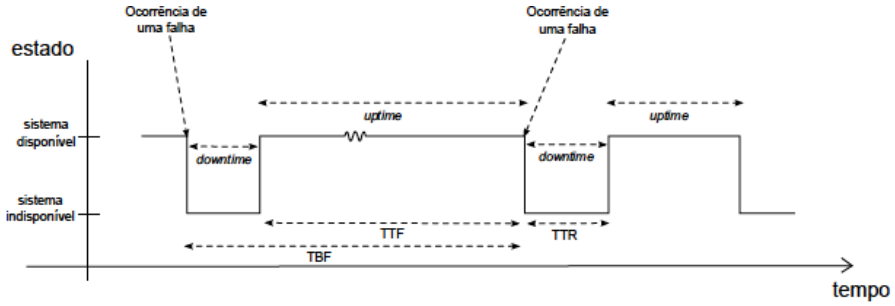

Figura 5 – *Uptime* e *Downtime* (MACIEL et al., 2011)

- **Tempo médio para falhas** - *Mean Time to Failure (MTTF)* - É o tempo médio para a ocorrência de falhas no sistema. Corresponde ao valor médio para a métrica TTF na figura 5 e é calculado pela Equação 2.6.

$$MTTF = \int_0^\infty R(t)dt \tag{2.6}$$

- **Tempo médio para reparo** - *Mean Time to Repair (MTTR))* – É o tempo médio para levar o sistema novamente ao estado de funcionamento, após a ocorrência de uma falha. Corresponde ao valor médio para a métrica TTR na Figura 5 e é calculado pela Equação

$$MTTR = MTTF \times (\frac{UA}{A}), \tag{2.7}$$

- **Tempo médio entre falhas** - *Mean Time Between Failures (MTBF)* – É o tempo médio entre a ocorrência de falhas. Corresponde ao valor médio para a métrica TBF da Figura 5.

A equação para o cálculo da disponibilidade pode ser escrita em função do **MTTF** e do **MTTR**, conforme apresentado na Equação 2.8.

$$A = \frac{MTTF}{MTTF + MTTR} \tag{2.8}$$

Quando o MTTF é muito maior que o MTTR, a disponibilidade pode ser avaliada de acordo com a Equação 2.9.

$$A = \frac{MTBF}{MTBF + MTTR} \tag{2.9}$$

2.4 Modelos de Avaliação

Os modelos de avaliação permitem representar relações mais complexas entre os componentes do sistema, como dependências que envolvem sub-sistemas e restrições de recursos. Modelos de dependabilidade são classificados em duas categorias: modelos combinatoriais e modelos baseados em espaço de estados (MACIEL et al., 2012).

Modelos combinatoriais capturam as condições que levam um sistema a falhar ou permanecer em funcionamento em termos de relacionamentos estruturais entre seus componentes. Exemplos de modelos combinatoriais são Diagramas de blocos de confiabilidade (*Reliability Block Diagram* (RBD)) e Árvores de falhas. Já os modelos Modelos baseados em estados, descrevem o comportamento do sistema em termos de estados e ocorrências de eventos, expressas como as transições de estado marcados, como Cadeias de Markov (*Continuous Time Markov Chain* (CTMC)) e Rede de Petri são dois tipos principais de modelos baseado em estados.

Esses tipos de modelos divergem em dois aspectos: sua facilidade em ser usado em aplicações específicas e o respectivo poder de modelagem. Esta suposição de independência pode ser muito restritiva e, desta forma, limitando o poder de expressividade desta classe de modelos. Modelos baseados em estados são mais complexos e difíceis de criar, os referidos possuem maior poder de expressividade do que modelos combinatoriais. Nesta seção apresentaremos os conceitos básicos do modelo utilizado neste estudo.

2.4.1 Rede de Petri

A representação gráfica das Redes de Petri é formada por lugares (Figura 6(a)), transições (Figura 6(b)), arcos (Figura 6(c)) e *tokens* (Figura 6(d)). Os lugares equivalem às variáveis de estado e as transições correspondem às ações realizadas pelo sistema (MACIEL, Paulo R M; LINS, Rafael D;CUNHA, 1996). Esses dois componentes são ligados entre si através de arcos dirigidos. Os arcos podem ser únicos ou múltiplos. A demanda de *tokens* (marcas) nos lugares da Rede de Petri determinam o estado do sistema ou a quantidade de recursos.

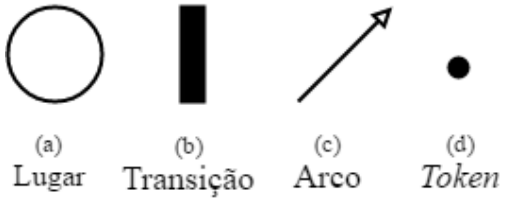

Figura 6 – Elementos de uma Rede de Petri

Na Figura 7, é apresentado um modelo de uma Rede de Petri que representa o *dia* e a *noite* a qual os *lugares* representam os períodos do *dia* e *noite*, já as transições alteram o período do dia (*amanhecer* ou *anoitecer*). Nesse exemplo, o arco dirigido do lugar *dia* para a transição *anoitecer* indica que, para que *anoiteça*, é necessário que haja um *token* no lugar *dia*. Tal qual, o arco dirigido do lugar noite para a transição amanhecer indica que, para que amanheça, é necessário que haja um *token* no lugar *noite*. A localização do *token* na rede indicará se é *dia* (Figura 7(a)) ou *noite* (Figura 7(b)).

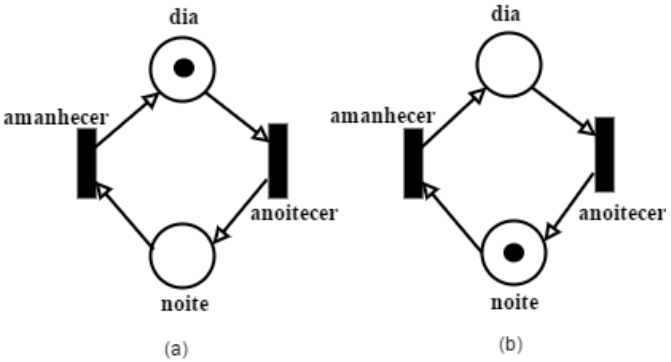

Figura 7 – Exemplo de uma Rede de Petri

2.4.2 Rede de Petri Estocástica - SPN

Redes de Petri (Petri Net - PN) são uma família de formalismos baseados em estado, apropriada para modelar diversos tipos de sistemas que possuam mecanismos de concorrência,assincronicidade, distribuição, determinísticos ou estocásticos (BALBO, 1989).

As redes de Petri estocásticas são uma extensão da rede de Petri e permitem a modelagem e análise probabilística de sistemas. São resultante das SPNs generalizadas (*Generalized Stochastic Petri Net (GSPN)*) (MARSAN; CONTE; BALBO, 1984), entretanto o acrônimo SPN (*Stochastic Petri Nets* (SPN)) é frequentemente usado para representar toda a família de modelos derivados da SPN, Em um modelo SPN (BALBO, 2001) o componente lugar é um componente passivo, uma variável do ambiente; o componente transição é um componente ativo, executa a ação do modelo utilizando os componentes de arco, que interligam e possibilitam a transição e mudança de lugar; estas transições podem ser imediatas ou temporizadas.

As atividades que possuem tempos associados, são representadas por transições temporizadas, as mesmas são identificadas por retângulos brancos (Figura 8 (a)). Nas SPNs o período de habilitação da transição corresponde ao tempo para realizar a atividade, e o disparo ao fim da atividade. As transições imediatas não têm tempo associado, e possuem prioridade de disparo maior que as transições temporizadas. Podendo também possuir prioridades e probabilidades entre transições imediatas diferentes. Os arcos inibidores (Figura 8 (b)), permite testar se um lugar não possui *tokens*. Com a presença do arco a transição estará habilitada se a quantidade de *tokens* em um lugar p associado ao arco for menor que o peso do arco n, ou seja $\mathbf{M(p) < n}$ (MARSAN et al., 1994).

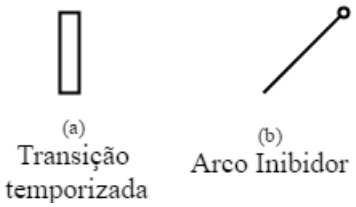

Figura 8 – Elementos adicionais de uma Rede de Petri

Adotaremos a definição formal de SPNs segundo (GERMAN, 2000), que é apresentada a seguir: Uma SPN é definida pela 9-tupla SPN = (P,T,I,O,H,$\prod$,G,M_0,Atts), onde:

- $\mathbf{P} = (p_1,p_2,...,p_n)$ é o conjunto de lugares. n é a quantidade de lugares;
- $\mathbf{T} = (t_1,t_2,...,t_m)$ é o conjunto de transições imediatas e temporizadas, $P \cup T = \emptyset$. m é a quantidade de transições;
- $\mathbf{I} \in (\mathbb{N}^n \longrightarrow \mathbb{N})^{n \times m}$ é a matriz que representa os arcos de entrada (que podem ser dependentes de marcações);
- $\mathbf{O} \in (\mathbb{N}^n \longrightarrow \mathbb{N})^{n \times m}$ m é a matriz que representa os arcos de saída (que podem ser dependentes de marcações);
- $\mathbf{H} \in (\mathbb{N}^n \longrightarrow \mathbb{N})^{n \times m}$ é a matriz que representa os arcos de inibidores (que podem ser dependentes de marcações);
- $\prod \in \mathbb{N}^n$ é o vetor que associa o nível de prioridade a cada transição;
- $\mathbf{G} \in (\mathbb{N}^n \longrightarrow \mathit{true}, \mathit{false})^n$ é o vetor que associa uma condição de guarda relacionada à marcação do lugar a cada transição;
- $\mathbf{Mo} \in (\mathbb{N})^n$ é o vetor que associa uma marcação inicial de cada lugar (estado inicial);
- $\mathbf{Atts} = (\mathit{Dist}; \mathit{Policy}; \mathit{Concurrency}; W)^m$ compreende o conjunto de atributos associados às transições:

Dist $\in (\mathbb{N})^m \longrightarrow F$ é uma função de distribuição de probabilidade associada ao tempo de cada transição, sendo que F $\leq \infty$. Esta distribuição pode ser dependente de marcação;

Policy $\in$ {*prd, prs*} define a política de memória adotada pela transição (*prd-preemptive repeat different*, valor padrão, de significado idêntico à *enabling memory policy; prs-preemptive* resume, corresponde a *age memory policy*);

Concurrency $\in$ {*ss,is*} é o grau de concorrência das transições, onde ss representa a semântica *single server*, e *is* representa a semântica *infinite server*;

$W \in (\mathbb{N})^+$ é a função peso, que associa um peso (w_t) às transições imediatas e uma taxa λ_t às transições temporizadas.

A adição de transições temporizadas introduz o conceito de habilitação múltipla, que deve ser considerado em transições temporizadas com grau de habilitação maior que um. Nesse caso a semântica de disparo deverá levar em conta a quantidade de *tokens* que podem ser disparados paralelamente. As possibilidades de semântica são:

- *Single Server - SS*: O tempo de disparo é contado quando a transição é habilitada,após o disparo da transição um novo tempo será contado se a transição ainda estiver habilitada. Portanto, os disparos ocorrerão em série, independente do grau de habilitação da transição;

- *Infinite Server - IS*: Todo conjunto de *tokens* da transição habilitada é processado simultaneamente. Então todos os *tokens* serão processados em paralelo;

- *Multiple Server*: Todo o conjunto de *tokens* será processado em paralelo até o máximo grau de paralelismo (k) definido para essa semântica.

As SPNs marcadas com um número finito de lugares e transições são isomórficas às cadeias de Markov (MARSAN et al., 1994). O isomorfismo de um modelo SPN com uma cadeia de Markov é obtido a partir do gráfico de alcançabilidade reduzido, que é dado através da eliminação dos estados voláteis e do rótulo dos arcos com as taxas das transições temporizadas e dos pesos das transições imediatas

Os modelos SPN descrevem as atividades de sistemas através de gráficos de alcançabilidade. Esses gráficos podem ser convertidos em modelos Markovianos, que são utilizados para avaliação quantitativa do sistema analisado. As medições de desempenho e dependabilidade são obtidas através de simulações e de análises em estado estacionário e transiente baseadas na cadeia de Markov embutida no modelo SPN (BOLCH et al., 1998).

2.5 Injeção de falhas

Quando trabalhamos com experimentos que detenham como objetivo a avaliação de dependabilidade, umas das técnicas que é frequentemente usada, denomina-se de injeção

de falhas. Tendo em vista que as mesmas ocorrem de forma imprevisível no sistema, esses eventos podem levar muito tempo para serem descoberto. A utilização de técnicas de injeção de falha auxiliam no controle e monitoramento do experimento, observando o comportamento do sistema durante a ocorrência de eventos de falha (ARLAT et al., 1990).

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, iniciam-se ações para conter a falha. Caso contrário, o sistema eventualmente apresenta mau funcionamento (ZIADE et al., 2004).

Para entender melhor o funcionamente de uma ferramenta de injeção de falhas, a Figura 9 define uma arquitetura básica para desenvolvimento de ferramentas de geração de eventos, esta estrutura foi baseada nos atributos de dependabilidade existentes. Onde: O *Controller* é o núcleo que detém toda a ferramenta, interligado a ele temos o *Monitor* que armazena os dados da injeção de falha, o *Fault injection* é o injetor que inicia os disparos no sistema a medida que executa ações enviadas pelo *Workload Generator* e o Monitor que analisa e coleta os dados provenientes da injeção de falhas.

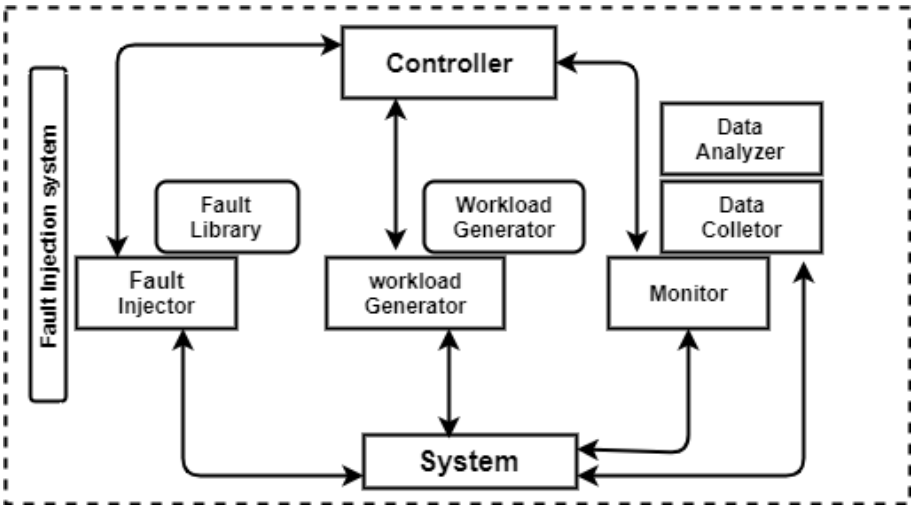

Figura 9 – Arquitetura de Injeção de falhas - Adaptado de (OLIVEIRA, 2014)

Na técnica de injeção de falhas podemos considerar que existem dois tipos de implementação: a nível de *software* e a nível de *hardware* (ZIADE et al., 2004). As falhas de *hardware* podem ser de dois tipos: com contato e sem contato. Na injeção de falha em *hardware* com contato, o injetor tem contato físico direto com o sistema alvo e pode produzir variações de corrente ou tensão no sistema testado. Na abordagem sem contato, o injetor não se conecta com o sistema alvo. Neste caso, para produzir as falhas, este produz algum evento externo que acarreta a falha do componente desejado (HSUEH; TSAI; IYER, 1997).

As técnicas de injeção de falhas por *software* também são divididas em duas categorias. A primeira é em tempo de compilação ou em tempo de execução. Injetores de falha

em tempo de compilação fazem alterações do código fonte do sistema original, a fim de emular o efeito de falhas de hardware e software no sistema. Injetores de falhas de tempo de execução, por sua vez, não exigem a modificação do código fonte do sistema. Em vez disso, este tipo de injetor utiliza algum mecanismo que serve de gatilho para a injeção de falhas. Existem três técnicas para implementar o gatilho que irá disparar as falhas (HSUEH; TSAI; IYER, 1997):

- **Time-out:** Esta técnica utiliza um *time*r (que pode ser de *hardware* ou de *software*) para controlar a emissão de falhas. Quando o contador do *timer* chega até zero, uma falha é injetada no sistema.
- **Exceção/trap:** Nesta técnica, quando uma certa exceção de *software* ou interrupção de *hardware* ocorre, o controle é transferido para o injetor de falhas. Esta técnica permite a injeção de falhas após a ocorrência de eventos específicos, coisa que a injeção por *time-out* não permite.
- **Inserção de código:** Esta técnica funciona adicionando código durante o tempo de execução do programa. As instruções originais não são modificadas, em vez disso, o código de injeção de falhas é adicionado após certos pontos do programa em memória principal.

A injeção de falhas no *hardware* exige a aquisição equipamentos custosos, enquanto que a injeção de falhas em software não possui essa exigência, tornando-se assim menos onerosa. Além disso, existe o risco de uma danificação permanente do dispositivo alvo, algo que não ocorre com a injeção de falhas em *software* (ZIADE et al., 2004).

Injetores em nível de *hardware* possuem um número limitado de pontos de injeção das falhas, enquanto que os injetores de *software* possuem um número de pontos de injeção mais amplo. Outra vantagem dos injetores de *hardware* em comparação com os de *software*, é que os de *hardware* não provocam perturbação no sistema alvo, enquanto que a perturbação provocada pelos injetores de *software* pode ser significativa e prejudicar os experimentos (OLIVEIRA, 2014). Sendo assim, conclui-se que a injeção de falhas por *software* deverá ser a primeira opção em testes, uma vez que a mesma é menos custosa e mais flexível.

2.6 Considerações Finais

Neste capítulo nós apresentamos os conceitos fundamentais necessários para a compreensão deste trabalho. Inicialmente abordamos sobre computação em nuvem, como surgiu e os serviços no qual a mesma oferece aos usuários, em seguida conceitos de *framework*. Logo após tratamos sobre a dependabilidade e suas ramificações, tal como a disponibilidade, conceito utilizado neste trabalho.

Em seguida abordamos sobre modelos de avaliação no qual utilizamos a rede Petri (PN), sua ramificação a SPN e por fim tratamos da injeção de falhas.

3 MODELANDO DISPONIBILIDADE COM MECANISMOS DE REDUNDÂNCIA ATRAVÉS DE SPN

Os modelos utilizados para avaliação de dependabilidade podem ser classificados como: Combinatoriais e baseados em Espaço de estados. Os modelos combinatoriais comumente utilizados são: Diagrama de bloco de confiabilidade (RBD - *Reliability Block Diagram*) e árvore de falhas (FT - *Fault Tree*), porém o RBD é o mais frequente. Modelos baseados em espaço de estados, representam o comportamento do sistema (ocorrência de falhas e reparo) por meio dos seus estados e da ocorrência de eventos. Os referidos modelos permitem a representação de relações de dependência entre os componentes do sistema, no qual podemos citar a rede de Petri estocásticas (SPN - Stochastic Petri Net) (BALBO, 2001). Visto que modelos SPN proporcionam grande flexibilidade na representação de aspectos de dependabilidade. Neste capítulo iremos explanar sobre os tipos de mecanismos de redundância existentes para o estudo de disponibilidade por meio de SPN.

3.1 Sistema Simplex

Os sistemas convencionais que não possuem redundância, são denominados de sistemas simplex, nesse tipo de sistema existe uma única unidade operacional disponível para esse serviço. O sistema possui dois estados operacionais: o primeiro chamado de *Working* ou "*UP*"assim caracterizado por estar em operacionalidade, disponível. O segundo denominado de *Failed* ou "*DOWN*", caracterizado por não estar operacional ou melhor indisponível (BAUER; ADAMS; EUSTACE, 2011). Além dos estados "*UP*"e "*DOWN*"existe um terceiro estado no qual podemos designar de sistema não-operacional, dado que o sistema presumi que o mesmo está operacional. Neste terceiro estado, o sistema não está entregando um serviço aceitável ou talvez esteja inoperante, porém este problema não foi detectado pelo usuário, em consequência, a recuperação automática, reparo ou substituição do componente não é efetuada (BAUER; ADAMS; EUSTACE, 2011).

Alguns trabalhos intitulam esta situação de falha como "falha descoberta", uma vez que a falha não foi detectada como deveria. Este estado de falha desconhecido às vezes é chamado de "falha silenciosa"ou "falha do sono", visto que o sistema não identificou automaticamente o problema ou não manifestou explicitamente o fracasso ao usuário.

Na Figura 10, consta um modelo SPN que categoriza o modelo simplex. A rede funciona da seguinte forma: o *token* no *place Componente_On*, indica que o sistema está disponível; em uma provável ocorrência de falha, a transição *MTTF_Componente* irá

disparar movendo o *token* do *place Componente_On* e/ou reparado *Componente_Off*, denotando que o sistema está inativo. Se esta falha for detectada (troca de componentes e etc), a transição *MTTR_Componente* é disparada movendo o *token* do *Componente_Off* para o *Componente_On*, tornando o sistema ativo novamente.

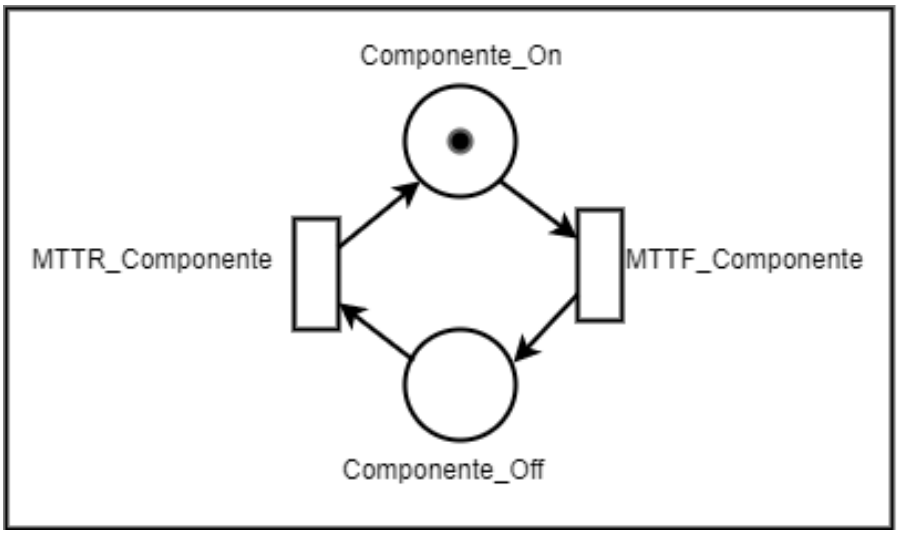

Figura 10 – Modelo SPN do sistema Simplex

O modelo matemático utilizado para estimar o tempo gasto em cada estado, pode ser encontrado com base em estimativas de taxa de falha, reparação e latência de recuperação. A probabilidade da falha ser detectada automaticamente e o tempo para encontrar uma falha (silenciosa) não transcorre de forma ágil. Obtemos esta formula em 2.3. Na Tabela 2, encontra-se os atributos que correspondem as transições do modelo Simplex (SOUSA, 2015).

Tabela 2 – Atributos das Transições - Modelo Simplex

Transição	**Tipo**	**tempo**	**Peso**	**Prioridade**	**Concorrência**
MTTF_Componente	exp	X_{MTTF}	-	-	SS
MTTR_Componente	exp	X_{MTTR}	-	-	SS

3.2 Mecanismos de Redundância

Os mecanismos de redundância proporcionam maior disponibilidade e confiabilidade ao sistema durante a ocorrência de eventos de falhas. Isso ocorre devido a manutenção de componentes operando em paralelo, portanto, um sistema redundante possuí um componente secundário que estará disponível quando o componente primário falhar. Assim, os mecanismos de redundância detém o objetivo de evitar pontos únicos de falha e consequentemente, proporcionar alta disponibilidade e recuperação de desastres, se necessário

(SOUSA, 2015) (BAUER; ADAMS; EUSTACE, 2011). Existem mecanismos de redundância denominados ativo-ativo e ativo-espera, no mecanismo ativo espera temos o *hot-standby*, cold-standby e *warm-standby*.

3.3 Mecanismos de redundância ativo-ativo

Esta classe de redundância é utilizada quando os componentes primários e secundários na ocorrência de qualquer falha de um dos componentes, o outro componente será o responsável por atender às requisições dos usuários do sistema. Portanto, classificamos este mecanismos de redundância como **N+K**, onde **K** componentes secundários idênticos aos **N** componentes primários, necessários para o compartilhamento da carga de trabalho do sistema. Sendo assim, na configuração **N+1**, um componente secundário idêntico aos **N** componentes primários é necessário para o compartilhamento da carga de trabalho do sistema. Em uma configuração **N+2**, dois componentes secundários idênticos aos **N** componentes primários fazem-se necessários para o compartilhamento da carga de trabalho do sistema (BAUER; ADAMS; EUSTACE, 2011), (TRIVEDI, 2008) e (SOUSA; MOREIRA; MACHADO, 2011).

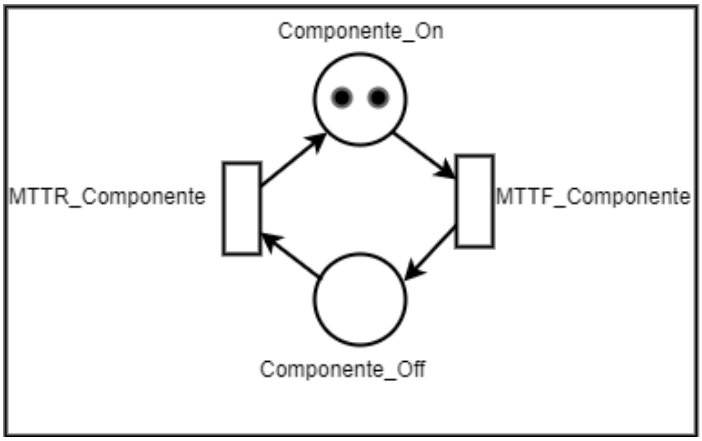

Figura 11 – SPN de Modelo ativo-ativo

Na Figura 11 conclui-se que: dois *tokens* no *place Componente_On*, indicam que o sistema possui duas máquinas funcionais, quando a transição *MTTF_Componente* é disparada, um *token* ocupa o lugar *Componente_Off* denotando a inatividade deste componente, entretanto, este sistema não torna-se indisponível, pois existe outro componente ativo na rede. Quando a transição *MTTR_Componente* é disparada quer dizer que o componente que estava com defeito, foi substituído. Por sua vez, o serviço não deixa de ser provido, como ocorre no sistema Simplex.

Na Tabela 3 encontramos os atributos da transição do modelo ativo-ativo. O sistema estará disponível caso o *Componente_On* seja menor que 1, Tabela 4.

Tabela 3 – Atributos das Transições - Modelo ativo-ativo

Transição	**Tipo**	**tempo**	**Peso**	**Prioridade**	**Concorrência**
MTTF_Componente	exp	X_{MTTF}	-	-	ISS
MTTR_Componente	exp	X_{MTTR}	-	-	SSS
ComponenteeReparo	im	-	1	1	-

Tabela 4 – Disponibilidade do modelo

Métrica	**Expressão**
Disponibilidade	$DP_{at} : P(\{\#Componente\_On > 1\})$

3.4 Mecanismos de redundância ativo-espera

São empregados quando os componentes primários atendem às requisições dos usuários do sistema e os componentes secundários estão em modo de espera. Quando os componentes primários falharem, os componentes secundários serão responsáveis pelo atendimento às requisições dos usuários do sistema. Os mecanismos de redundância ativo-espera podem ser classificados como: *hot standby*, *cold standby* e *warm standby*.

3.4.1 Hot Standby

Em uma redundância *hot standby* os dados dos sistemas são mantidos em sincronia com a outra maquina da rede, portanto ao surgir uma ocorrência de falha, uma das máquinas deve ser rapidamente capaz de assumir o serviço, uma vez que seu outro componente que estava ativo falhou. Todos os dados, sejam importantes ou não devem ser replicados automaticamente a partir do momento que a máquina que falhou for recuperada (BAUER; ADAMS; EUSTACE, 2011).

Em *hot standby*, a rede pode ser ativada automaticamente ou manualmente. Normalmente, os componentes enviam mensagens periodicamente para os componentes da rede, tanto os que estão em espera, quanto os que estão ativos. Por isso quando umas das unidades detecta o mal funcionamento da outra, logo assume-se o serviço (BAUER; ADAMS; EUSTACE, 2011). Desta forma, a principal característica de um componente que utiliza redundância *hot standby* é a ausência de tempo de ativação (se comparado com as redundâncias *cold standby* e *warm standby* (SOUSA, 2015).

O Modelo *hot standby* é similar ao modelo ativo-ativo, porém a diferença é que em caso de ocorrência de falha, a máquina assume o serviço da outra (SOUSA, 2015). A Figura 12 mostra o modelo SPN adotado para estimar a disponibilidade de um componente. Observa-se que: dois *tokens* no *place Componente_On*, indicam que o sistema possui duas máquinas funcionais, quando a transição *MTTF_Componente* é disparada, um *token* ocupa o lugar *Componente_Off* denotando a inatividade deste componente, entretanto, este sistema não torna-se indisponível, pois existe outro componente ativo na rede. Quando

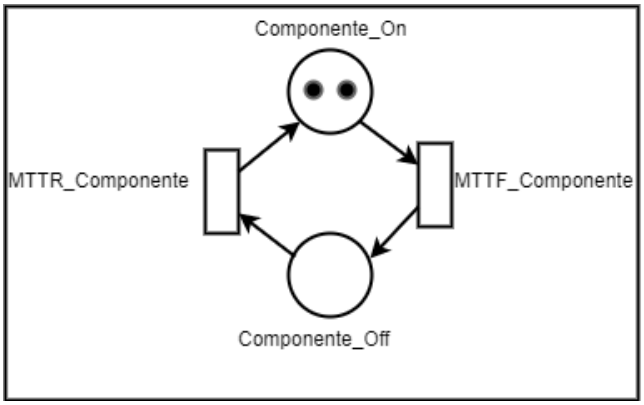

Figura 12 – SPN do modelo *hot standby*

a transição *MTTR_Componente* é disparada quer dizer que o componente que estava com defeito, foi substituído. Entende-se que este componente não está indisponível, em razão da existência de outro componente ativo na rede.

A Tabela 5, apresenta os atributos das transições do modelo SPN *hot standby*. Por conseguinte na Tabela 5 encontra-as a formula da disponibilidade que diz: O sistema estará disponível caso o *Componente_On* seja menor que 1.

Tabela 5 – Atributos das Transições - Modelo *Hot Standby*

Transição	**Tipo**	**tempo**	**Peso**	**Prioridade**	**Concorrência**
MTTF_Componente	exp	X_{MTTF}	-	-	ISS
MTTR_Componente	exp	X_{MTTR}	-	-	SSS
ComponenteeReparo	im	-	1	1	-

Tabela 6 – Disponibilidade do modelo

Métrica	**Expressão**
Disponibilidade	$DP_{hs} : P(\{\#Componente\_On > 1\})$

3.4.2 Cold standby

Em uma redundância *cold standby* o componente de redundância não está ativo. Para ser restaurado um serviço de uma unidade ativa com falha, o modo de espera deve ser ligado, o *Sistema Operacional* (SO) deve ser iniciado, e os dados devem ser carregados a partir de uma cópia de *backup*. Este procedimento pode levar algumas horas para completar-se, a isso intitulamos de *Mean Time To Activate (MTTA)*. Indicando que o ajuste por vezes pode ser manual (BAUER; ADAMS; EUSTACE, 2011) (SOUSA, 2015).

Em outras palavras, um componente com redundância *cold standby* é baseado em um módulo redundante não ativo, que por sua vez, espera para ser ativado quando o módulo

principal ativo falha. Dessa forma, quando o módulo principal falha, a ativação do módulo redundante ocorre em um certo período de tempo (SOUSA, 2015) Abaixo podemos ver um modelo em SPN de um sistema *cold standby*, através da Figura 13.

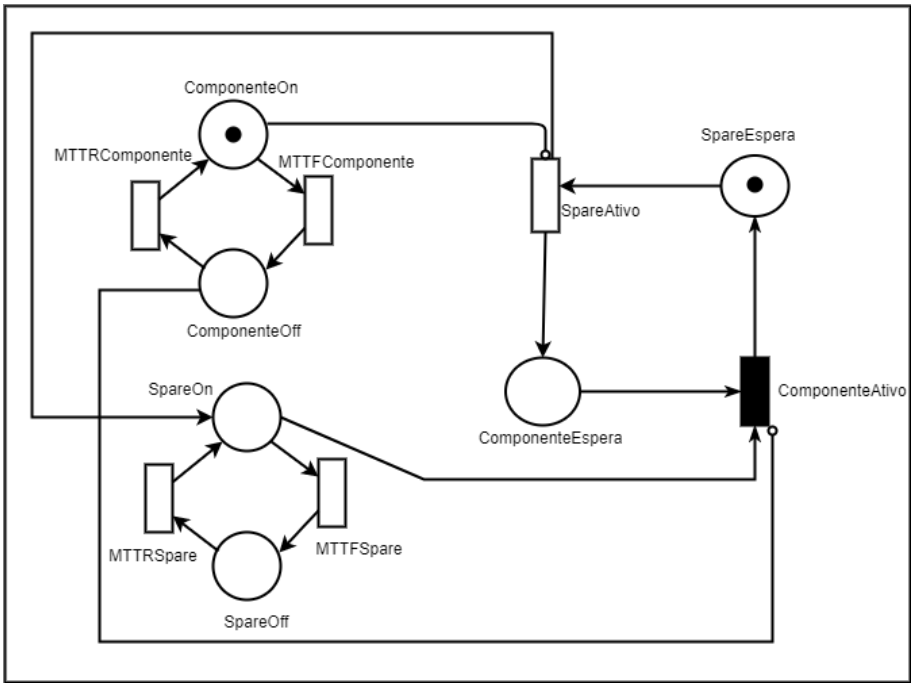

Figura 13 – Modelo SPN *cold standby*

Iremos mostrar como exemplo de uma SPN para estimar a disponibilidade de um componente com redundância *cold standby*, um modelo adotado por (SOUSA, 2015). As marcações dos lugares *ComponenteON*, *SpareON*, *ComponenteOFF* e *SpareOFF*, exibem os estados operacionais e de possíveis falhas de ambos os módulos principal e redundante, respectivamente. O módulo redundante está inicialmente desativado, uma vez que não exista *tokens* nos lugares *SpareON* e *SpareOFF*. Quando o módulo principal falha, a transição temporizada *SpareAtivo* é disparada. O tempo associado à transição temporizada *SpareAtivo* faz menção ao *Mean Time to Active* (MTA), uma marcação no lugar *SpareEspera* corresponde ao módulo reserva, que por sua vez, não está operacional. As transições temporizadas *MTTFComponente*, *MTTFSpare*, *MTTRComponente* e *MTTRSpare* representam a ocorrência de um evento de falha e de uma atividade de reparo nos módulos principal e redundante, e os tempos associados a essas transições temporizadas representam o *MTTF* e o *MTTR* desses componentes.

A Tabela 7 detém os atributos relacionados as transições. Enquanto naa Tabela 8, observa-se a formula utilizada para calcular a disponibilidade do sistema. E pode ser entendida como: a infraestrutura de nuvem estará em modo operacional quando houver pelo menos uma marcação nos lugares *Componente_On* ou no *Spare_On*.

Tabela 7 – Atributos das Transições - Modelo *Cold Standby*

Transição	**Tipo**	**tempo**	**Peso**	**Prioridade**	**Concorrência**
MTTF_Componente	exp	MTTFC	-	-	SS
MTTF_Componente	exp	MTTRC	-	-	SS
MTTF_Spare	exp	MTTFS	-	-	SS
MTTR_Spare	exp	MTTRS	-	-	SS
SpareAtivo	exp	MTA	-	-	SS
ComponenteAtivo	im	-	1	1	-

Tabela 8 – Disponibilidade do modelo

Métrica	**Expressão**
Disponibilidade	$DP_{cs} : (P\{(\#Componente\_On = 1 OR \#Spare\_On = 1)\}$

3.4.3 Warm standby

Um componente que possui redundância *warm standby* baseia-se em um módulo redundante não-ativo, ou seja, que espera para ser ativado quando o módulo principal ativo falha. A diferença entre à redundância *cold standby* e a *warm standby*, é que o módulo principal e o módulo redundante possuem uma taxa de falhas λ quando estão em operação, mas o módulo redundante possui uma taxa de falha 0 quando encontra-se desenergizado (BAUER; ADAMS; EUSTACE, 2011).

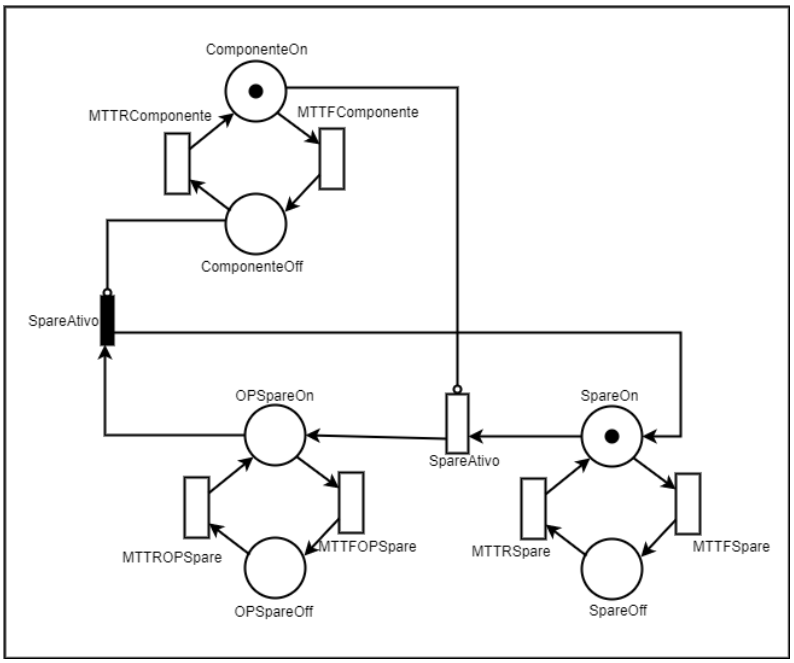

Figura 14 – Modelo SPN *Warm standby*

Utilizamos o modelo *warm standby* de (GUIMARÃES; MACIEL; MATIAS, 2013), para compreensão do seu funcionamento. Na Figura 14, observa-se que a referida possui seis

lugares, os lugares do módulo principal *ComponenteON* e *ComponenteOFF* e os lugares do componente redundante. Esses lugares do componente redundante representam o seu estado de atividade e de inatividade. Os lugares *SpareON* e *SpareOFF* representam o componente reserva em modo inativo. Já os lugares *OPSpareON* e *OPSpareOFF* representam o componente reserva em modo ativo. O componente redundante começa em modo inativo. Quando o módulo principal falha, a transição temporizada *SpareAtivo* dispara. Esse disparo representa o inicio da operação do componente redundante. O tempo associado à transição temporizada *SpareAtivo* representa o *Mean Time to Active* (MTA). Já a transição imediata *SpareNAtivo* representa o retorno do componente principal para o modo operacional. As transições *MTTFComponente*, *MTTFSpare*, *MTTFOPSpare*, *MTTRComponente*, *MTTRSpare* e *MTTROPSpare* e representam a ocorrência de um evento de falha e de uma atividade de reparo dos módulos principal e reserva. Na Tabela 9, podemos observar os atributos das transições deste modelo, e na Tabela 10, podemos ver a formula da disponibilidade deste sistema que diz: Este sistema computacional estará em modo operacional quando houver pelo menos uma marcação nos lugares *Componente_ON* ou *OPSpare_ON*.

Tabela 9 – Atributos das Transições - Modelo *Warm Standby*

Transição	**Tipo**	**tempo**	**Peso**	**Prioridade**	**Concorrência**
MTTF_Componente	exp	MTTFC	-	-	SS
MTTF_Componente	exp	MTTRC	-	-	SS
MTTF_Spare	exp	MTTFS	-	-	SS
MTTR_Spare	exp	MTTRS	-	-	SS
MTTF_OPSpare	exp	MTTFOPS	-	-	SS
MTTR_OPSpare	exp	MTTROPS	-	-	SS
SpareAtivo	exp	MTA	-	-	SS
SpareNAtivo	im	-	1	1	-

Tabela 10 – Disponibilidade do modelo

Métrica	**Expressão**
Disponibilidade	$DP_{ws} : (P\{(\#Componente\_On = 1 OR \#OPSpare\_On = 1)\}$

3.5 Considerações Finais

Neste capítulo nós inicialmente apresentamos os tipos de modelos utilizados para avaliação de disponibilidade de sistemas. Que podemos caracterizar em: Modelos combinatoriais e baseado em espaços de estados. Os modelos combinatoriais comumente utilizados são o RBD e a arvóre de falha e nos baseados em estados a SPN. Além disso temos os

mecanismos de redundância, no qual temos o ativo-ativo e ativo-espera. O ativo-espera é dividido em: *hot standby, cold standby e warn standby.*

4 INJEÇÃO DE FALHAS ATRAVÉS DE SPNs E MONITORAÇÃO DE DISPONIBILIDADE EM NUVENS PRIVADAS

Para compreensão do *framework* desenvolvido neste trabalho, é importante entender como o mesmo foi idealizado e o seu diferencial com relação a outras ferramentas existentes na literatura. Neste Capítulo, abordaremos duas estrategias de injeção e monitoramento de falhas em plataformas de nuvens computacionais.

A primeira denominada de estrategia **Tradicional** é baseada em ferramentas que foram desenvolvidas para um determinado fim, e para que possam ser utilizadas em ou outros ambientes precisam ser modificadas. Esta modalidade de ferramental normalmente não oferece margem para analisar sistemas distintos do qual foi implementada, o que consequentemente impossibilita sua reutilização. A segunda a ser abordada neste trabalho, é apresentada como estrategia **Proposta**, onde não torna-se necessário aplicar modificações ao código da ferramenta para adequá-la ao ambiente de teste. Uma vez que a referida utiliza como mecanismo de injeção de falha modelo formal SPN, possibilitando ao usuário do *framework* apenas a mudança de modelo para análise de diferentes plataformas.

Fundamentados nesses princípios, podemos compreender a importância em desenvolver ambientes que proporcionem meios para avaliação e disponibilidade nos sistemas de nuvem computacional. Vale ressaltar que é de fundamental importância a utilização de ferramentas reutilizáveis e dessa forma evitando que os instrumentos caiam em desuso.

4.1 Injeção de Falhas Utilizando SPN

A cada instante são desenvolvidos ferramentas geradoras de eventos, a fim de validar e analisar o comportamento dos sistemas mediante a ação de carga de trabalho. Isto decorre em vista de que o pesquisador, modelou seu código baseado em um problema ou plataforma especifica.

Com relação a esta problemática, foi proposto desenvolver um ambiente que oferecesse aos pesquisadores a possibilidade de avaliar vários modelos de plataforma de nuvem, sem que houvesse a necessidade de desenvolver códigos, *scripts* ou outras técnicas dispendiosas para os mesmos. Foi utilizado o modelo formal SPN devido ser caracterizado como uma abstração da realidade, seu jogo simbólico torna possível a criação de ambientes virtuais nos quais são gerados modelos com uma relação causa-efeito, e assim podemos

comparar com um sistema real. Um modelo SPN possui *delay* em seus disparos e faz uso de uma probabilidade exponencial.

Figura 15 apresenta dois cenários: o primeiro apresentado como **Ambiente do usuário** e o segundo a plataforma de nuvem que intitulamos de **Cloud**. No primeiro ambiente encontra-se o usuário acessando sua máquina (*hardware*) e nesta máquina temos o sistema operacional (*software*) neste local situa-se a ferramenta de injeção de falhas e monitoramento.

A ferramenta de injeção de falhas desenvolvida é baseada em um ambiente especifico (plataforma, problemática, entre outros) caso o usuário queira utilizar esta ferramenta em um estudo diferente, o mesmo será obrigado a fazer uma modificação no código deste ferramental para se adequar ao ambiente no qual ele deseja avaliar. Isso não torna esta ferramenta inútil ela pode ser reutilizada, mas terá que ser modificada o que aumenta o retrabalho do pesquisador. Na Figura 15 ao observar esse ambiente entendemos que esta ferramenta precisará de uma atualização para que possa funcionar na avaliação de outros modelos.

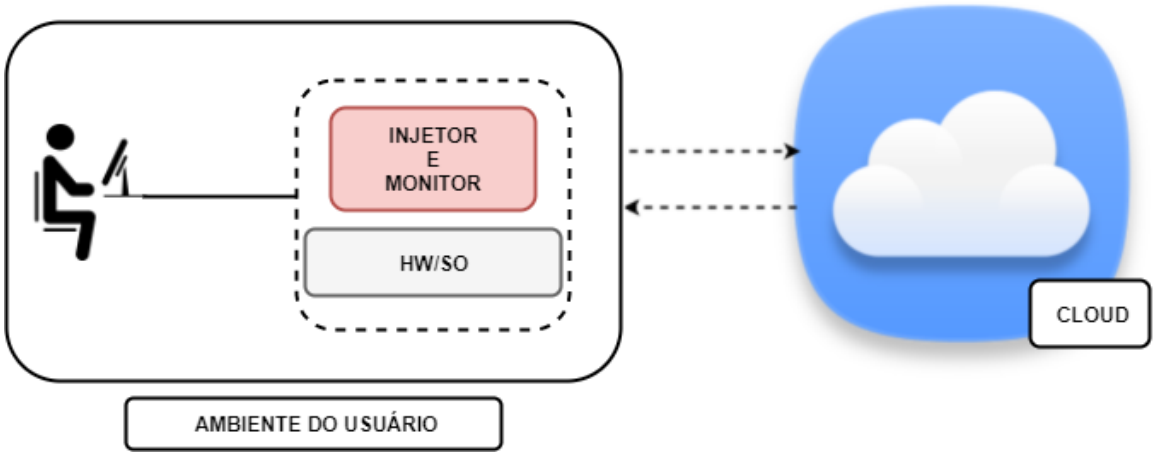

Figura 15 – Modelo de ambiente Tradicional

Figura 16 detalha um ambiente diferente do modelo da estrategia tradicional. Contando com dois cenários **Ambiente do usuário** e **Cloud**, observamos o pesquisador utilizando a máquina física (*Hardware* (HW)), que automaticamente possui um sistema operacional (*software*), além disso nesta máquina encontra-se uma ferramenta de modelagem de formalismos matemáticos, onde desenvolveu-se um modelo SPN para avaliar um sistema de nuvem computacional. Ao contrário da Figura 15, o ambiente proposto, possui acoplado ao seu injetor e monitor, um modelo SPN que será utilizado como um mecanismo de injeção de falha. Assim, o usuário poderá criar o modelo SPN e aplicá-lo dentro do ferramental concluindo o processo de validação do ambiente em um cenário real, sem a necessidade de modificar a ferramenta para alcançar seu proposito.

Não é necessário depurar o modelo SPN no código pois o ambiente no qual será avaliado não é parâmetro para codificação da ferramenta. Caso o pesquisador queira

analisar outra nuvem computacional, utilizando um modelo SPN distinto, basta retirar o mesmo da ferramenta, o que não ocasionará mudanças ao código, visto que a SPN é o mecanismo de injeção de falhas e reparos.

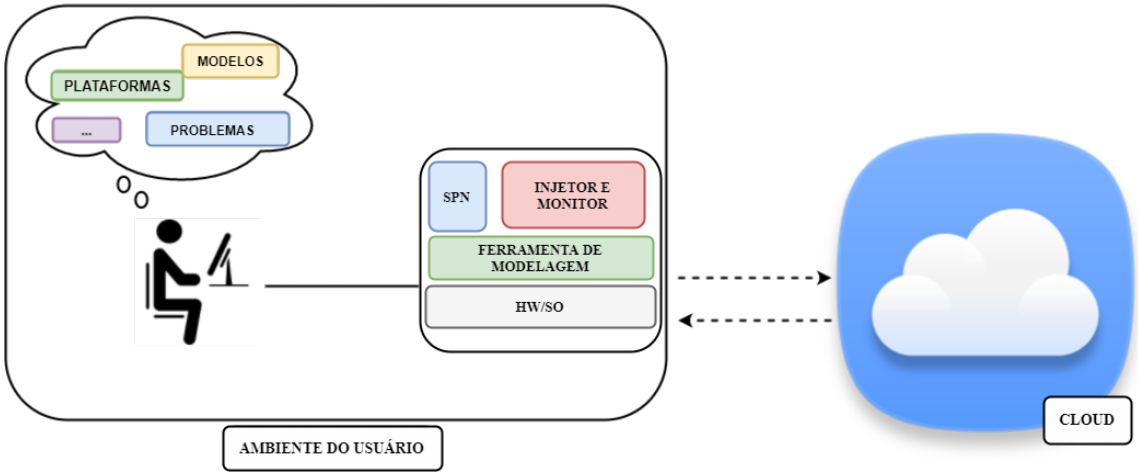

Figura 16 – Modelo de ambiente Proposto

Percebeu-se que a utilização da estrategia proposta irá contribuir para a diminuição do retrabalho por parte pesquisadores para avaliação de ambientes de nuvem.

4.1.1 Entendendo o funcionamento do Ambiente

Através da Figura 17, iremos exemplificar para melhor compreensão como desenvolveu-se o ambiente proposto.

Um pesquisador pretendia avaliar a disponibilidade de um ambiente de nuvem, o mesmo organiza uma plataforma de nuvem física, esta plataforma distinta é caracterizada por dois servidores: o primeiro é a máquina *frontend* (controlador da nuvem) e o segundo o *node* (componente responsável pela instanciação de máquinas virtuais). Abaixo dos servidores observa-se o modelo SPN onde: O *token* no **Frontend_On** indica que a máquina frontend está ativa, quando a transição **Frontend_Falha** é disparada o *token*, passa do estado On (ativo) para o estado **Frontend_Off**, que indica que esta maquina está inativa, depois de um certo período de tempo, a transição **Frontend_Repair** é ativada, reparando o sistema e o mesmo volta para a condição inicial de ativo, onde o *token* ocupa novante o lugar **Frontend_On**.

A mesma situação ocorre com a máquina **Node**, onde: O *token* no lugar **Node_On** indica que a máquina está ativa, quando a transição **Node_Falha** é disparada, o *token*, sai do estado ativo e é transferido para o estado **Node_Off**, que denota a inatividade da máquina, depois de algum tempo a maquina é reparada através da transição **Node_Repair**, e o *token*, volta para o lugar **Node_On**, demonstrando que a rede está ativa novamente.

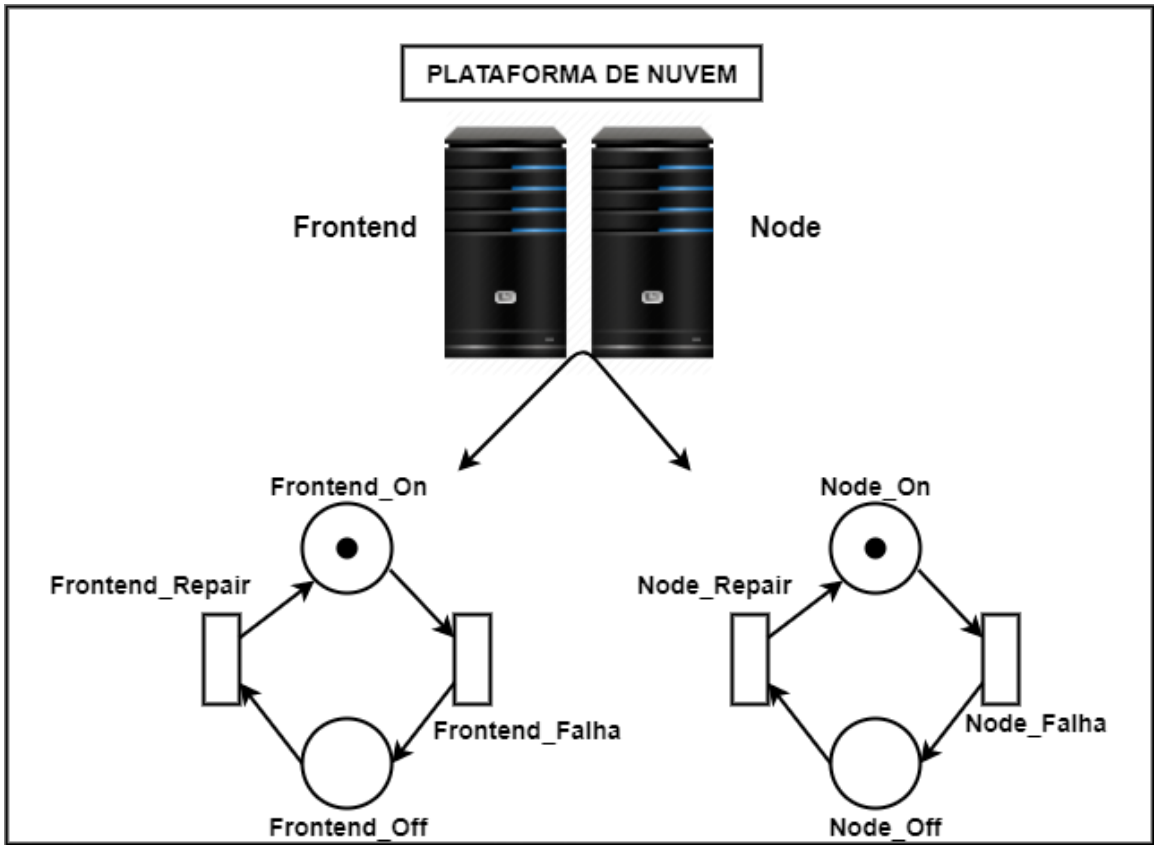

Figura 17 – Modelo de SPN de uma Plataforma de Nuvem

De posse dos dados de sua SPN, isto é, valores de falha e reparo das transições de ambos os componentes, observando também se a propriedade estrutural desta rede realmente estão corretas, o usuário de acordo com a Figura 18 irá salvar seu modelo SPN desenvolvido a partir de uma **ferramenta de modelos matemáticos** e depois irá salvar este modelo em formato de *script*, que será implantado dentro do ambiente de injeção e monitoramento de falhas. Antes de ser executado a ferramenta irá realizar a tradução do *script* do modelo que será utilizado como mecanismo de injeção de falhas na plataforma de nuvem analisada.

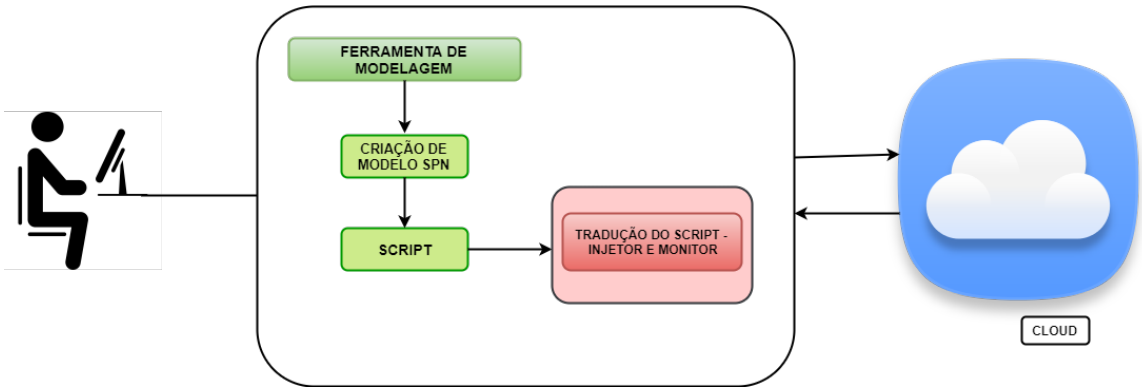

Figura 18 – SPN utilizada como mecanismo de uma Plataforma de Nuvem

As SPNs são definidas assumindo que todas as transições são temporizadas e que o atraso no disparo das transições é associado a uma variável aleatória distribuída exponencialmente. A Figura 19, mostra qual componente dispara as falhas e reparos na plataforma de nuvem, sendo assim, podemos observar na SPN, que a máquina frontend que possui a transição **Frontend_Falha** corresponde a ação de disparo de falha na rede e a transição **Frontend_Repair** está relacionada ao momento em que a rede é reparada. O mesmo acontece na máquina **Node_On**, os tempos de falha e reparo dessas máquinas, são exponenciais, esse tempo que está imbuído tanto na transição de Falha, quanto na transição de reparo, indica o momento no qual será executada as ações das transições, os disparos dentro desse ambiente ocorrem a partir de um *time out*. O *time out* nada mais é do que um contador que possui um tempo especifico para disparar suas ações, esse tempo é determinado dentro do ambiente de injeção de falha e associado as transições da SPN. O ambiente leva em consideração os estados de uma SPN, e a injeção ocorre por simulação, o usuário não saberá em qual componente iniciara a injeção de falha, pois a intenção é que a SPN comporte-se como um modelo real ao ser acoplada a ferramenta.

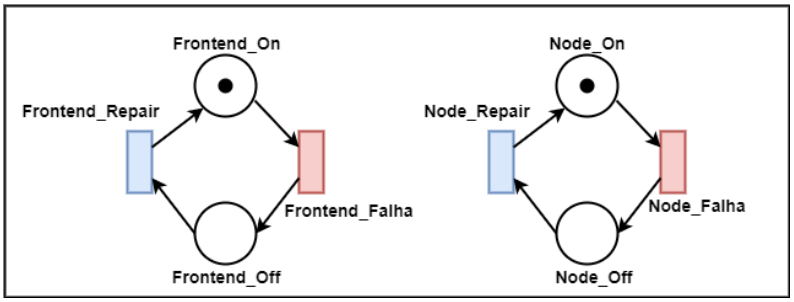

Figura 19 – Falha e Reparo - Transições

4.2 Monitoração de Disponibilidade em Nuvens Privadas

O Monitoramento de nuvem é uma tarefa de suma importância para ambos os provedores e consumidores. Por outro lado, é fundamental para o controle e gerenciamento de infraestruturas seja de *hardware* ou *software*, pois fornece informações sobre o desempenho das plataformas avaliadas. É importante que durante os processos que ocorrem dentro de uma plataforma, seja possível observar o andamento do sistema, se as falhas estão sendo injetadas, ora se o sistema está sendo reparado, ou até mesmo se os componentes da nuvem estão ativos antes da ocorrência.

Na Figura 20 cada lugar (place) da SPN equivale a um IP da rede, visto que antes de ser efetuado os disparos é indispensável que o programa entenda qual IP corresponde ao lugar da SPN. Ao ser informado o IP, o monitor realiza uma comunicação com o componente no qual a SPN irá disparar a transição de falha (**Frontend_Falha**), se a máquina está ativa, o monitor irá exibir ao usuário "**Frontend_On**: Conexão realizada com Sucesso!", senão o mesmo irá informar "**Frontend_On**: Conexão não Realizada!". Quando a conexão é aceita inicia-se o disparo das transições, o monitor informa se o componente do ambiente sofreu uma falha através do evento "Down", essa palavra é exibida ao usuário cada vez que uma falha ocorre, além da data e hora que o sistema parou de funcionar.

Depois de um tempo, através do *time out*, é disparada uma ação de reparo deste componente, através da transição **Frontend_Repair**, quando essa ação ocorre, o monitor informa em sua interface, o momento da ocorrência e em seguida exibe o evento "Up"que significa que o sistema está ativo e funcionando, tal como na ocorrência do evento "Down", no evento "Up"é informado também a data e hora que o sistema está ativo. Logo após o reparo e observação do sistema, a simulação SPN é disparada em outro componente da rede, o **Node_On**. Na máquina **Node_On** ocorre o mesmo processo de monitoramento do **Frontend_On**, entretanto a SPN já entende qual IP pertence a cada máquina na rede, o monitor inicia a verificação do Node, e emite a mensagem "**Node_On**: Conexão não Realizada!", após essa mensagem, inicia-se a primeira ocorrência de falha, que provem da transição **Node_Falha**, logo após é exibida a mensagem "Down"que denota que o sistema está inativo. Após a falha, a transição **Node_Repair** é acionada, recuperando o sistema e tornando ativo novamente, todas as informações são exibidas pelo o monitor em tempo real, e ao final do processo o mesmo emitira um relatório com as informações para o usuário.

Apresentamos o relacionamento do monitor se relaciona com o ambiente proposto:

- **Injetor SPN**: É criado um modelo SPN pelo pesquisador em um ambiente de modelagem, onde o modelo será exportado em formato de script, o mesmo será acoplado dentro do injetor, onde será traduzido e fará comunicação com o núcleo gerador de números aleatórios. A injeção de falha acontece através de simulação, os disparos nos devidos componentes ocorrem de forma aleatória, não seguindo uma

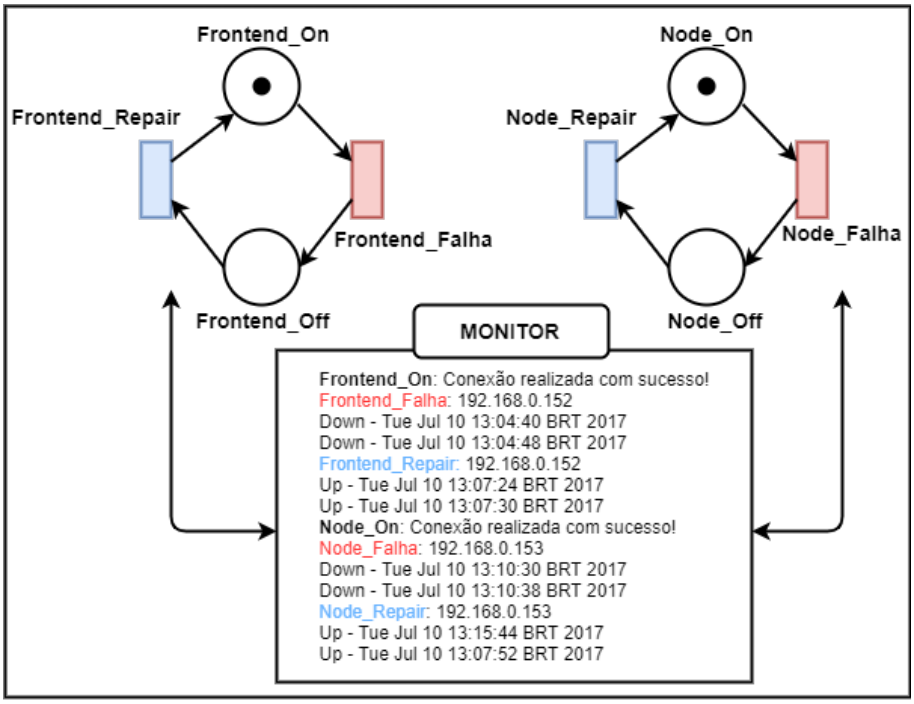

Figura 20 – Processo de Monitoramento do Ambiente

cadeia linear (*frontend* e *node*, mais *node* ou *frontend* e assim sucessivamente). O tempo de falha (MTTF) e o tempo de reparo (MTTR) são agregados dentro do injetor de falhas. O gerador de números aleatórios, que carrega a probabilidade exponencial, está junto as transições e reponde pela contagem do tempo, fazendo com que as mesmas obedeçam o valor agregado ao sistema.

- **Monitor**: Trabalha em conjunto com as transições de falha e reparo da SPN, monitorando a disponibilidade do sistema, verificando cada mensagem recebida e avaliando a integridade da mesma. Se a mensagem contem erro, ou se o sistema não respondeu da maneira esperada, ainda assim a mensagem é exibida, para que o pesquisador tenha ciência do que está acontecendo dentro do sistema. Ao final é gerado um relatório, com todas as ocorrências do sistema durante o período de injeção de falha.

4.2.1 Percepção do Ambiente

Uma vez que o modelo SPN esteja bem implementado, ou seja, o usuário observou que pelos métodos matemáticos o mesmo retrata com fidelidade o sistema e sua estrutura não apresenta problemas. Inicia-se o processo de injeção de falhas, caso o pesquisador esteja necessitando analisar outro modelo basta retirar o *script* SPN do ambiente. Quando o modelo SPN é submetido a um cenário de teste, o ferramental possui a capacidade de verificar a rede através do monitor.

Por fim, entende-se que no ambiente Tradicional a ferramenta é desenvolvida utilizando uma problemática especifica (modelos, *softwares*, plataformas e etc.) a codificação deste ambiente ocorreu de uma ideia particular. Para que esse *software* possa ser utilizado para um fim dissemelhante ao qual foi proposto, o usuário terá que fazer uma atualização do código tornando-se um trabalho dispendioso e suscetível a erros.

Por sua vez, o ambiente Proposto usufrui do formalismo SPN como um mecanismo de injeção de falhas, e mesmo modificando o modelo SPN o mesmo pode ser reutilizado. A SPN não possui ligação com o código do *framework*, ou seja, não serviu como parâmetro para criação deste ambiente facilitando a avaliação de outras infraestruturas de nuvem. Esta estratégia foi desenvolvido devido a necessidade de ferramentas de carga de trabalho para plataformas de computação em nuvem, que oferecessem suporte para auxiliar os pesquisadores na avaliação a disponibilidades de sistemas, visto que valida-se um modelo formal juntamente com um ambiente real.

Foi observado que em diversos trabalhos as ferramentas geradas possuíam limitações com relação a sua continuidade, ora fosse por motivos de *software* ou de *hardware*. Percebeu-se que grande parte eram desenvolvidas para sanar um problema ou para uma plataforma computacional especificas restringindo o pesquisador de analisar outros sistemas e negligenciando a mesma. A realização desta incumbência 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. Acredita-se que este ambiente, irá conceber a facilidade de avaliar outros sistemas de plataforma de nuvem ou modelos. No próximo Capítulo apresentaremos o funcionamento estrutural deste ambiente.

4.3 Considerações Finais

Esse capítulo apresentou os conceitos para o entendimento do *framework* apresentado neste trabalho. Dividiu-se a análise em dois pontos: ambiente tradicional e ambiente proposto, o ambiente tradicional pode ser entendido como uma ferramenta que precisará de atualizações para que possa ser utilizadas em ambientes diferentes no qual foi desenvolvida e o ambiente proposto é entendido como uma ferramenta que utiliza SPN como um mecanismo para injetar falhas em um plataformas de nuvem e assim, não é necessário modificar o código para adapta-lo a outras plataformas.

5 SIMF:FRAMEWORK DE INJEÇÃO DE FALHA E MONITORAMENTO PARA CLOUD UTILIZANDO SPN

Este Capítulo apresenta o *framework* de injeção de falha e monitoramento denominada **SIMF**. O mesmo possui subsídios para compreensão do desenvolvimento da estrategia proposta pelo Capítulo 4. Especificamente esta ferramenta foi desenvolvida para avaliação de disponibilidade de sistemas de nuvem computacional. Seu diferencial é beneficiar-se do modelo formal SPN, utilizando-o como mecanismo de injeção de falha.

5.1 Visão Geral

O *framework* SIMF (*SPN Injection and Monitoring of Failures Framework*) é um gerador de eventos de falhas com opção de reparo, que emula e simula a ausência de operações no sistema de nuvem gerenciado por plataformas computacionais. O diferencial desta ferramenta é a aplicação do formalismo matemático rede de Petri estocástica (SPN), como um mecanismo que dispara falhas e reparos na plataforma.

As falhas causadas pelo SIMF são transientes, ou seja, o referido simula um possível estado de interrupção da execução do sistema, de modo que o mesmo possa ser reparado. As perturbações causadas através da inserção de falhas no sistema de nuvem, buscam afetar a execução dos componentes de alto-nível da plataforma de nuvem. A ação de reparo objetiva recuperar o sistema de eventuais falhas causadas pelo injetor.

O SIMF possui as seguintes operações:

- **Tradução SPN**: Utiliza a ferramenta geradora de modelos analíticos *Mercury*, para criar um modelo SPN, salva este modelo em formato script. O mesmo é traduzido dentro do SIMF, que utiliza a simulação *tokengame* (jogo simbólico para análise de modelo estocástico) do *Mercury*, para injetar falha na nuvem.

- **Falhas e reparos de software**: SIMF injeta falhas em componentes de plataformas de nuvens (*Frontend*, Nó entre outros de escolha do usuário) ou pode ser falhas de *hardawre*. A ferramenta atua diretamente, suspendendo a execução de um processo selecionado pelo usuário. Vale ressaltar que na mesma máquina pode-se injetar mais de um tipo de falha de software. O reparo é realizado através da restauração dos processos que foram anteriormente finalizados pelo SIMF.

- **Monitor**: O monitor possui a finalidade de checar o *status* da rede e informar ao usuário se os serviços estão ocorrendo da maneira devida, ao final exibe um relatório do que ocorreu durante o processo de injeção de falhas. Além disso, o mesmo processa o tempo de atividade e inatividade da rede, esta ocorrência é observada em tempo real.

O SIMF possibilita que o processo de injeção ocorra segundo a SPN desenvolvida o *framework* não é restrito a um único modelo criado pelo usuário. Visto que entendemos que cada sistema comporta-se ou é estabelecido de maneira diferente, o mesmo ocorre com a SPN. Atualmente o *framework* restringe-se a ambientes que utilizem o sistema operacional Linux, no entanto caso o usuário queira utilizar outra plataforma de nuvem, basta realizar algumas modificações com o que se diz respeito a parâmetros de falha e reparo. Neste caso, o desenvolvedor pode adicionar outras falhas de software. O manual do usuário contendo maiores informações sobre como utilizar o SIMF pode ser visto no Apêndice B.

Figura 21 expressa um usuário manuseando SIMF inicialmente é concebido o modelo SPN que é transferido para o *framework* onde será empregado como mecanismo de injeção de falha. Na imagem entende-se que o modelo SPN pode ser criado a partir de uma problemática especifica.

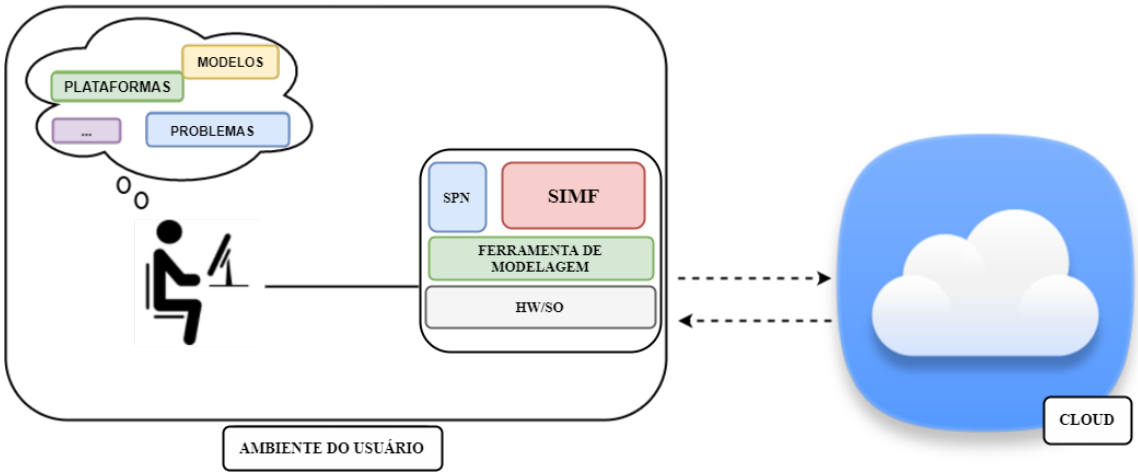

Figura 21 – *Framework* SIMF

5.1.1 Kernel SIMF

Foi utilizado o *Kernel* da ferramenta [1]EucaBomber para composição deste *framework*, dentre todas as distribuições distintas na ferramenta EucaBomber, a distribuição exponencial foi escolhida devido seus valores estarem mais próximos da realidade, além de ser aplicada ao modelo SPN em suas transições de falha e reparo.

[1] Ferramenta de injeção de falhas para nuvem *Eucalyptus*

De forma sucinta, o *kernel* oferece apenas o meio de comunicação com o sistema alvo e o gerenciamento da ocorrência de eventos. Segundo (SOUZA, 2013), 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 Linux presente nos servidores que compõem a nuvem. O módulo de geração de números aleatórios é responsável por gerar números pseudo-aleatórios, a distribuições de probabilidade utilizada no *framework* SIMF foi a exponencial A ferramenta utiliza estes valores como intervalo de tempo (*time out*) para a ocorrência dos eventos. É importante ressaltar que este *kernel* foi previamente testado.

SIMF pode injetar mais de uma falha por componente ao mesmo tempo ou seja , é possível analisar o sistema por inteiro, isso ocorre em virtude da realização do o "jogo simbólico"(simulação) que o *framework* possui. Figura 22, ilustra os componentes utilizados do EucaBomber para compor o *framework*. O diagrama de classe do EucaBomber encontra-se no Apêndice C.

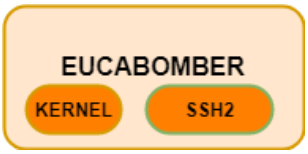

Figura 22 – Componentes Utilizados do EucaBomber

5.1.2 Script

Para obter métricas de avaliação de dependabilidade ou desempenho de sistemas em geral, é preciso o apoio de ferramentas de *software*. A ferramenta *Mercury*, que permite a criação e avaliação de modelos: cadeias de *markov* (CTMC), diagrama de blocos de confiabilidade (RBD), Modelos de fluxo de Energia (EFM) e Rede de Petri Estocástica (SPN). O Mercury fornece uma interface gráfica para o usuário, além de permitir também a integração com aplicativos externos. O *Mercury* apresentado nesta pesquisa foi desenvolvido pelo grupo de pesquisa [2]MODCS, essa ferramenta possui mais de 25 distribuições de probabilidade. Sabemos que existem muitas ferramentas de software acadêmico e comercial de avaliação de sistemas. O *Mercury* foi usado em vários projetos no qual seus resultados foram publicados em revistas. E novas versões são lançadas a cada seis meses aproximadamente.

Para auxiliar o usuário na avaliação de modelos SPN, o *Mercury* possui um jogo simbólico que é chamado de *token game*, que torna possível o disparo das transições

[2] Modeling of Distributed and Concurrent Systems

graficamente, os disparos iniciam-se de acordo com a marcação atual da Rede de Petri estocástica e assim os usuários podem avaliar as propriedades estruturais da SPN. Este ferramental é útil para academia e indústria, pois auxilia no planejamento de capacidade de diferentes sistemas por meio de avaliação de desempenho, previsão de confiabilidade através da dependabilidade e avaliação de disponibilidade.

Entretanto, apesar de sua robustez não é possível executar uma simulação de injeção de falha em um cenário real de uma plataforma de nuvem. Pensando nisso, decidiu-se utilizar a robustez desta ferramenta, para desenvolver um *framework* de injeção de falha para nuvem, que agregasse um modelo formal, a fim de validar o mesmo em experimentos. A partir do *Mercury*, deu-se inicio a implementação do SIMF. Utilizamos a simulação *token game* para efetuar os disparos de falha e reparo na plataforma de nuvem, e levamos em consideração todos os componentes que fazem parte da SPN, como *arcos*, transições, *places* entre outros.

É necessário enfatizar que a injeção de falha, agregando SPN não é gráfica, os acontecimentos se devem por tradução do *script*, outra informação relevante, é que o SIMF é dependente do *Mercury*, pois precisa de um modelo SPN desenvolvido por essa ferramenta para que a injeção de falhas seja promovida. Utilizou-se desta ferramenta as classe de simulação de *token game* e da SPN do *Mercury*.

5.2 Descrição SIMF

Para o desenvolvimento do SIMF adotamos algumas escolhas sendo elas: i); ii) protocolo SSH2 e iii) ferramenta geradora de modelos que seria agregada ao sistema, a fim de ser unificada ao injetor de falhas.

O diferencial do SIMF está na sua integração com a rede de Petri (SPN), o usuário pode injetar falha em uma infraestrutura de nuvem qualquer utilizando o modelo SPN. Antes de iniciar a injeção de falhas usando um modelo SPN, é importante salientar que o usuário faça uso da ferramenta *Mercury*, uma vez que a injeção de falhas ocorre a partir de um modelo SPN desenvolvido por ela. Ao gerar o modelo SPN, é necessário inserir o referido no injetor, que antes do processo oferece a oportunidade de abrir o arquivo desejado onde foi salvo.

No modelo SPN, cada *place* está associado a um IP (Internet Protocol) de uma máquina de infraestrutura de nuvem. Antes da injeção de falha, o monitor inicia o processo de comunicação com a mesma, verificando se cada componente está disponível e em seguida, inicia o processo de injeção de falhas. Neste trabalho, decidimos inserir e reparar falhas, os reparos são selecionados no sistema alvo em tempo de execução. Para desencadear ambos os eventos usamos um mecanismo de tempo limite, que está acoplado as transições da SPN. Entendemos que esse mecanismo é mais apropriado porque o testador apenas precisa inserir seus parâmetros (por exemplo: valores de falha e reparo) o modo

de execução da ferramenta estará no Apêndice B.

Por conseguinte, os eventos gerados utilizam o *kernel* para emular o tempo entre falhas, o tempo entre esses eventos foi provocado com base no gerador de números aleatórios que usa a distribuição exponencial. As unidades de tempo para a ocorrência dos eventos estão disponíveis em meses, dias, horas, minutos, segundos ou milissegundos. A unidade de tempo usada pelo injetor de falha foi de milissegundos e a distribuição de probabilidade utilizada foi exponencial. O tempo é importante, porque as variáveis aleatórias são usadas como um atraso entre um estado do sistema e outro até o próximo evento ser disparado. As falhas são independentes uma da outra; A única dependência existente entre uma falha é sua ação de reparo. O SIMF emula e simula uma injeção de falhas em um ambiente real.

O monitor do *framework* é responsável por supervisionar os serviços da plataforma de nuvem computacional e informar ao usuário através de mensagens se as falhas estão ocorrendo ou não no sistema, a fim de que o pesquisador possa analisar e observar o comportamento da nuvem. Estas informações facilitam o entendimento das condições do ambiente, além de informações sobre possíveis erros. O monitor informa: O componente da nuvem juntamente com o IP e o *place* da SPN no qual está ocorrendo a falha; as ocorrências de falha e reparo, contabilizando-as e a data. Além de informar o momento de atividade (UP) e inatividade (Down) do sistema. O monitor possui um papel importante que é efetuar a situação do sistema e dessa forma, apresenta todos os resultados obtidos a partir de uma interface. Isto significa que a aplicação possui a capacidade de verificar constantemente o estado das máquinas presentes na rede, como também os respectivos serviços da nuvem.

5.3 Diagrama UML

A UML (*Unified Modeling Language*) é uma linguagem visual utilizada para modelar softwares baseados no paradigma de orientação a objeto. Podemos classificar a referida como uma modelagem de proposito geral com a habilidade de ser agregada a todos os domínios da aplicação. Esta linguagem tornou-se padrão, visto que é adotada internacionalmente pela industria de engenharia de software. A necessidade de modelar um software, surge em decorrência da documentação extremamente detalhada que o sistema precisa ao ser concebido, dessa forma a modelagem é uma forma eficiente de documentá-lo (GUEDES, 2008).

Dentre os diagramas UML existentes utilizou-se apenas dois: o diagrama de classes é uma representação gráfica estruturada que descreve as classes e seus relacionamentos, e o diagrama de sequência que retrata as interações entre os objetos.

5.3.1 Diagrama de classe

O diagrama de classe é o mais utilizado tornando-se um dos mais importantes da UML. Serve como apoio para a maioria dos demais diagramas, como o próprio nome diz, define a estrutura de classes utilizadas pelo sistema. Determinando os atributos e métodos que cada classe possui, além de estabelecer como as mesmas relacionam-se e trocam informações entre si (GUEDES, 2008).

No diagrama de classe correspondente a Figura 24, temos as classes: **TokengameTest**, **EventsTest**, **Injector**, **Mercury** e **Monitor**. Classes com funções semelhantes foram agrupadas no mesmo pacote, na qual chamamos de "**Interface**", onde encontram-se as interfaces gráficas apresentadas sequencialmente ao usuário a medida que os dados solicitados em cada uma delas vão sendo preenchidos. Através de herança são criadas classes especializadas com atributos e métodos necessários para a interface gráfica. Na classe **Interface**, encontramos o atributo *IP* onde o usuário especifica quais IPs pertencem a cada componente da rede; o *Password* é a senha de acesso ao sistema; *Script* é o modelo SPN para ser traduzido; *Time* é tempo de duração do teste (1 dia, 2 dias e entre outros); *Status* é a confirmação de que as informações inseridas estão corretas e que a injeção de falhas irá acontecer, caso falte alguma informação (rede SPN errada, máquinas não estão ativas e entre outros), o sistema retornará um *Status* de falha.

A **classe Mercury**, aplica-se ao modelo SPN que será manipulado para injetar falha, ao finalizar a modelagem dentro da *classe Mercury*, os parâmetros de transição e lugar da SPN são invocados, para que seja efetivada a tradução do *script* (*.mry*). A *classe Mercury* contém a *classe TokengameTest*, a mesma está interligada a essa classe, visto que a *classe TokengameTest* inexiste sem a *classe Mercury*. Esta classe é responsável pela tradução do *script*, exibindo as matrizes que representam o modelo SPN. Em seguida, a *classe TokengameTest* envia esta informação para a *classe EventsTest*.

A *classe TokengameTest* corresponde ao "*Token Game*"que por sua vez, é decorrente de uma funcionalidade da ferramenta Mercury na qual os usuários simulam o comportamento dos modelos SPN. Para iniciar esta funcionalidade no Mercury os usuários devem clicar no botão "*Token Game*", e para ocorrer os disparos devem clicar duas vezes em uma transição ativa com o botão esquerdo do mouse, isso faz com que ele dispare alterando a marcação da rede. A diferença desta classe dentro do SIMF, dar-se em decorrência dos disparos das transições da SPN que ocorrem aleatoriamente, sem a necessidade de clicks do usuário para disparar os eventos, além do mais cada transição possui um valor estabelecido dentro do código, valores que são retirados de dentro do *kernel* do injetor (tempos de falha e reparo).

Após ser feita a tradução do script pela *classe TokengameTest*, a mesma repassa essas informações para a **classe EventsTest**, responsável pelo recebimento dos parâmetros das transições. Esta classe realiza a captura dessa tradução, transformando-a para linguagem Java, associa os IPs dos componentes da nuvem, onde cada *places* da SPN

possui um caminho e dispara a simulação dos componentes através do *simulacaoToken*. Após isso, fazendo uso da simulação realiza-se uma chamada a *classe injetor* para iniciar os disparos na nuvem, por exemplo; o primeiro evento disparado na simulação SPN, na tradução foi o **node2On**, dessa forma, este será o primeiro componente no qual ocorrerá a injeção de falha. A injeção, ocorre através dos disparos da simulação, *transitionFailure* e *transitionRepair*.

Na **classe injector** ocorre a injeção de falhas e reparos nos componentes da nuvem, essa classe possui o *kernel* responsável pelos números pseudo-randômicos, na qual atribui-se o *randVariateGeneration*, e a conexão SSH2 através do *sshCommand*. É importante ressaltar, que esses tempos de falha e reparo são atribuídos as transições do modelo SPN, pois os disparos ocorrem de acordo com a simulação do modelo. O *kernel* é o 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, esses valores são disparados levando em consideração a probabilidade exponencial. É na *classe injector* que verifica-se a inicialização por meio do *InicializarComponentes* da injeção de falha *transitionFailure* por meio do reparo do sistema *transitionRepair*. Além dos status verificando se os componentes estão ativos na rede com o uso do *isAlive*. Caso o modelo SPN criado pelo o usuário possua Máquinas Virtuais(VM) em sua estrutura, o injetor possui um método que verifica se as mesmas estão ativas, tanto na injeção de falha como no reparo, visto que a VM está em uma camada virtualizada e o processo para verificação é mais complexo do que o das máquinas físicas, os comandos são: *isVM* e *eucaCommand*.

A **classe monitor** é responsável por informar ao usuário o que ocorre na rede durante a injeção de falha, a mesma informa as ocorrências de falha e reparo dos componentes da nuvem, estas ações transcorrem em tempo real, ao final da injeção de falha transmite-se ao usuário um relatório com todas as ocorrências da nuvem durante o processo. Nesta classe encontramos o atributo *Components* que está relacionado aos componentes da nuvem (Frontend, Node e VM). O mesmo realiza a verificação destes módulos. O *temporizador* indica o tempo em que o monitor percorre cada componentes averiguando o status da rede, esta ação acontece a cada 6 segundos aproximadamente. O *geradatas* informa no final da injeção de falha os dados do sistema.

Na Figura 23 podemos entender o funcionamento do monitoramento do SIMF, selecionamos o momento em que a ferramenta está inspecionando o componente Node. Existe dentro do monitor parâmetros de checagem dos dados no campo *ResultUpNode.Up()*, onde armazena-se o tempo em que o Node está ativo; *ResultDownNode.Down()* informa que o mesmo está inativo; *ResultFailNode.fail()* armazena o instante que a falha é inserida no sistema e *ResultRepairNode.Repair()*. O sistema checa a infraestrutura a cada 6 segundos. A cada checagem SIMF armazena o instante em que o componente está ativo (UP) e o momento em que está inativo (DOWN), quando a falha é disparada (Falha) e quando o

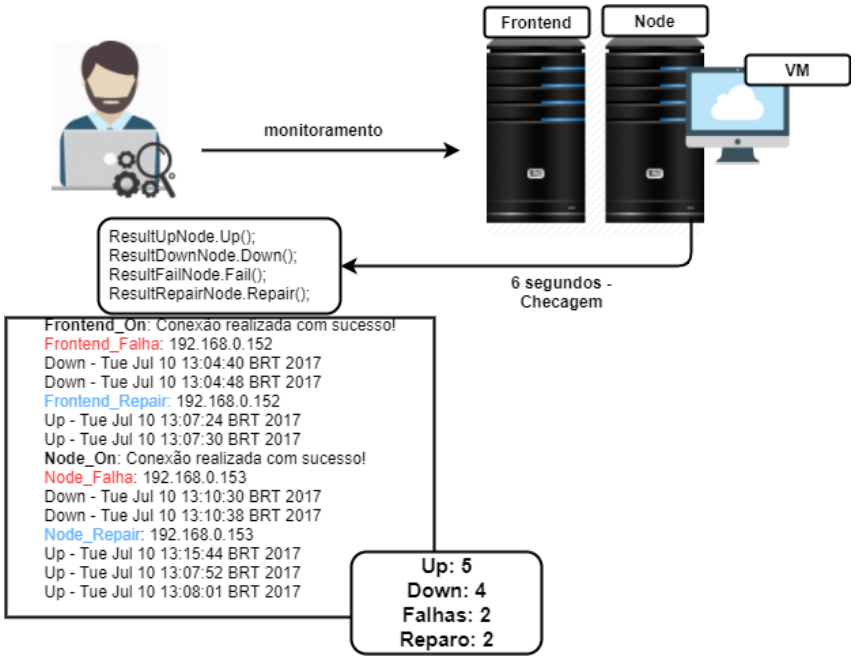

Figura 23 – Funcionamento do Monitor do SIMF

componente é reparado (Reparo).

5.3.2 Diagrama de Sequência

O diagrama de sequência é comportamental e preocupa-se com a ordem temporal em que as mensagens são trocadas entre os objetos envolvidos em um determinado processo. Em geral baseia-se em um caso de uso definido pelo diagrama de mesmo nome e apoia-se no diagrama de classes em um determinado processo (GUEDES, 2008).

No diagrama de sequência da Figura 25, observa-se que no primeiro retângulo no qual está o *User*, ou seja, corresponde ao usuário responsável por criar o Modelo SPN na ferramenta *Mercury*, em seguida, salvar em formato script, *saveScript*. A mensagem, *scriptSPN* denota o momento em que o script é enviado para o ambiente *SIMF*, onde a mensagem *translateScript*, informa que a ferramenta irá traduzir a mesmo para iniciar o processo de injeção de falha. Após o script ser traduzido, ocorre o evento de injeção de falha, onde é disparado o *transitionFault*, que envia o reparo para o *Cloud Components*, que por sua vez engloba todos os componentes da plataforma de nuvem. Se o sistema após a injeção de falha estiver inativo, o mesmo retornará ao sistema com a mensagem *Down*.

Quando o SIMF recebe a mensagem (Down) é disparado a *transitionRepair*, que corresponde ao reparo do componente, logo após é informado ao usuário que o sistema está (Up). Este (*Loop*) irá ocorrer enquanto durar o processo de injeção de falhas. O *alt* significa que o reparo ocorre quando a opção *down* é satisfeita. Como foi explicado anteriormente, dependendo do modelo SPN criado pelo usuário, o mesmo pode possuir uma VM, a referida é exibida no *CloudComponents*, as transições de falha e reparo estão

associadas a mesma, pois entende-se que esta faz parte dos componentes da nuvem. Ao final é possível constatar que o monitor que averiguá o status da rede e informa em tempo real para o usuário, ao fim da injeção entrega-se o arquivo (relatório) para análise dos dados obtidos. Assim podemos resumir:

- Utiliza modelo SPN para injetar falha e reparo em uma plataforma de nuvem distinta em tempo de execução. Durante a execução do sistema, as falhas e o reparo são injetados usando comandos de software (por exemplo, stop CLC);
- É um ambiente no qual o modelo não está depurado dentro do código e não está interligado a uma plataforma de nuvem;
- O SIMF possui um monitor que observa se o sistema está ativo ou inativo e calcula o tempo entre falhas (Down) e reparos (Up), este relógio registra o tempo de inatividade e o tempo de atividade de cada componente durante a execução do sistema;
- O tempo entre falhas e reparos é gerado pelo pacote EucaBomber, através do pacote FlexLoadGenerator. O tempo de duração da experimento é em Milissegundos e os valores adotados nos modelos SPN são os mesmos dentro do injetor de falhas.

5.4 Considerações Finais

Este capítulo apresentou o *framework* SIMF, como derivou-se e quais os mecanismos utilizados para seu funcionamento, descrevemos seus componentes abordando tópicos sobre o seu *kernel* e a geração do *script* SPN pela ferramenta *Mercury*. Foi possível observar seu surgimento e o modo como foi desenvolvida através dos diagramas de classe e sequência. Para utilização do *framework* temos um tutorial no Apêndice B.

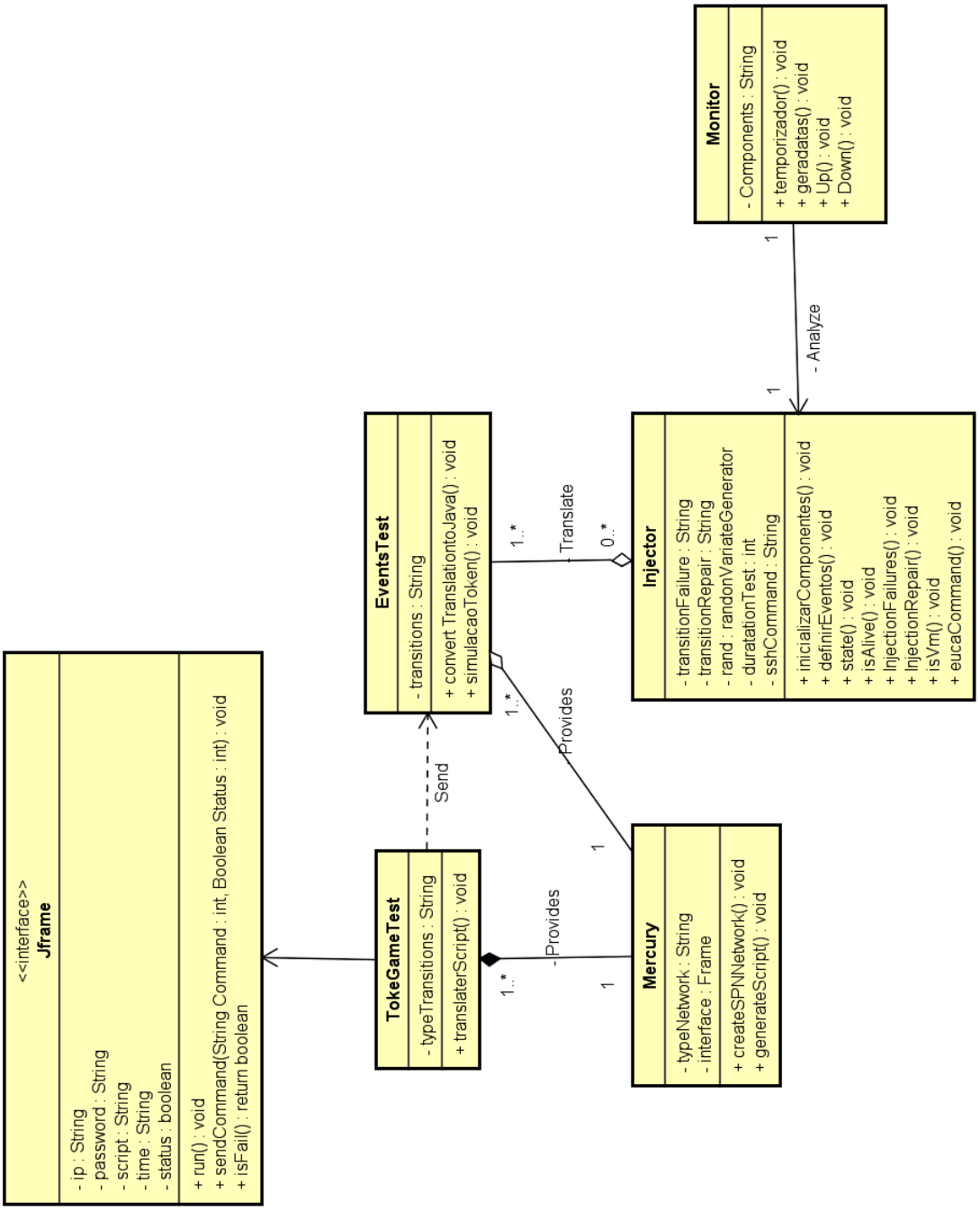

Figura 24 – Diagrama de Classe SIMF

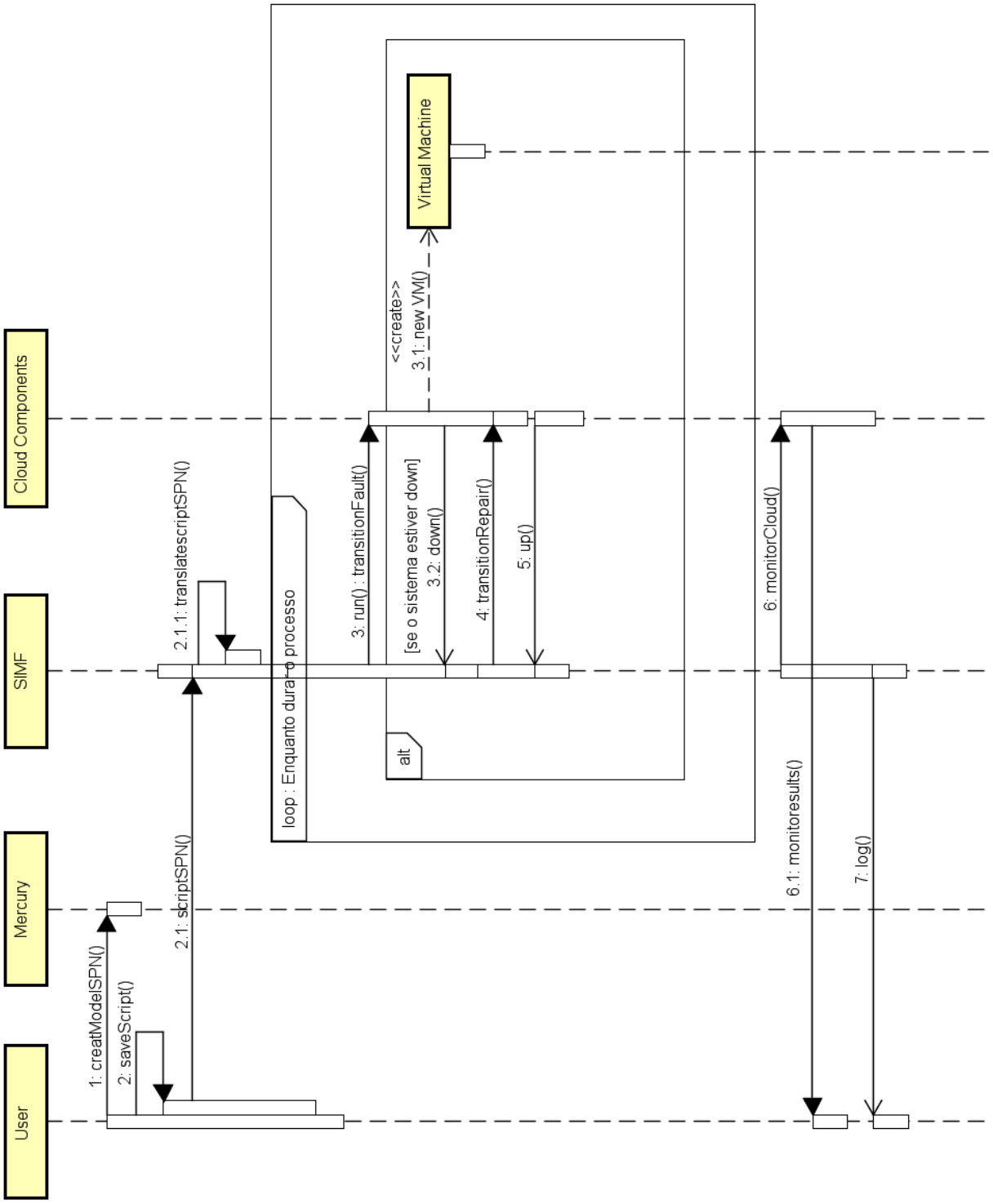

Figura 25 – Diagrama de Sequência SIMF

6 ESTUDO DE CASO

Neste capítulo iremos apresentar dois estudos de caso que foram validados utilizando o *framework* SIMF. O primeiro estudo de caso é validar o modelo SPN em uma plataforma de nuvem *Eucalyptus*,utilizamos a ferramenta SIMF, após esta verificação empregou-se a técnica denominada análise de sensibilidade, que diz a respeito da avaliação do componente de maior criticidade do modelo validado. Em consequência dos resultados destes dados do estudo de caso I, originou-se o segundo modelo concernindo no estudo de caso II.

Evidencia-se que o principal intuito deste capítulo é demostrar a eficiência da ferramenta geradora de eventos construída. Além disso, este estudo busca verificar se a disponibilidade do modelo SPN antes da validação coincide com a disponibilidade obtida no ambiente real ao manusear *framework*, e analisar o impacto de distintos MTTF (*Mean Time to Failure*) e MTTR (*Mean Time to Repair*) na disponibilidade do sistema em nuvem.

O *framework* SIMF foi desenvolvido para avaliar diversas plataformas de nuvens computacionais, utilizando o modelo SPN que o usuário pretende analisar em um ambiente real, porém nestes estudos foi utilizado somente a plataforma de nuvem *Eucalyptus*.

6.1 Estudo de Caso I

O propósito deste estudo de caso é validar o modelo SPN decorrente de uma plataforma de nuvem *Eucalyptus* utilizando a ferramenta SIMF.

Este estudo está estruturado da seguinte forma: primeiro denota-se a plataforma de nuvem *Eucalyptus*; Em seguida apresentaremos o cenário de testes tal como as configurações das máquinas físicas que compunham a plataforma; Logo após será apresentada a arquitetura e o modelo SPN da referida; Em seguida o processo de validação deste modelo comparando com a disponibilidade encontrada pela ferramenta SIMF e do modelo analítico, finalizando com a análise de sensibilidade.

6.1.1 Nuvem *Eucalyptus*

A infraestrutura da nuvem empregada neste cenário de teste é denominada de *Eucalyptus* (Elastic Utility Computing Architecture Linking Your Programs To Useful Systems) iremos abordar sobre sua plataforma e seus conceitos gerais.

A nuvem privada *Eucalyptus* é descrita como uma arquitetura de software baseada em Linux que implementa nuvens privadas e híbridas escaláveis dentro de sua infraestrutura de TI. É compatível com pacotes para múltiplas distribuições do *Linux*, incluindo o

Ubuntu, *RHEL*, *OpenSuse*, *Debian*, *Fedora*, e o *CentOS*, a mesma permite que o usuário possa manipula-la de forma simples. Esta plataforma foi desenvolvido com a finalidade de auxiliar na pesquisa com enfase na área de computação em nuvem, sua interface é compatível com o serviço comercial da *Amazon*, o *Amazon EC2* (JOHNSON et al., 2010a). A compatibilidade da API permite executar um aplicativo da *Amazon* no *Eucalyptus* sem modificação. No geral, a plataforma do *Eucalyptus* utiliza a capacidade de virtualização (*hypervisor*) do sistema de computador subjacente para permitir alocação flexível de recursos de computação dissociados de *hardware* específico (JOHNSON et al., 2010a).

A nuvem *Eucalyptus* é composta por vários componentes, que interagem uns com os outros através de uma interface bem definida. Há cinco componentes de alto nível da arquitetura *Eucalyptus* cada um com seu próprio *web service*: *Cloud Controller*, *Cluster Controller*, *Node Controller*, *Storage Controller*, e *Walrus* (EUCALYPTUS SYSTEMS, INC., 2009). Podemos observar na Figura 26 um exemplo de um ambiente base de computação em nuvem *Eucalyptus*, considerando dois clusters (A e B).

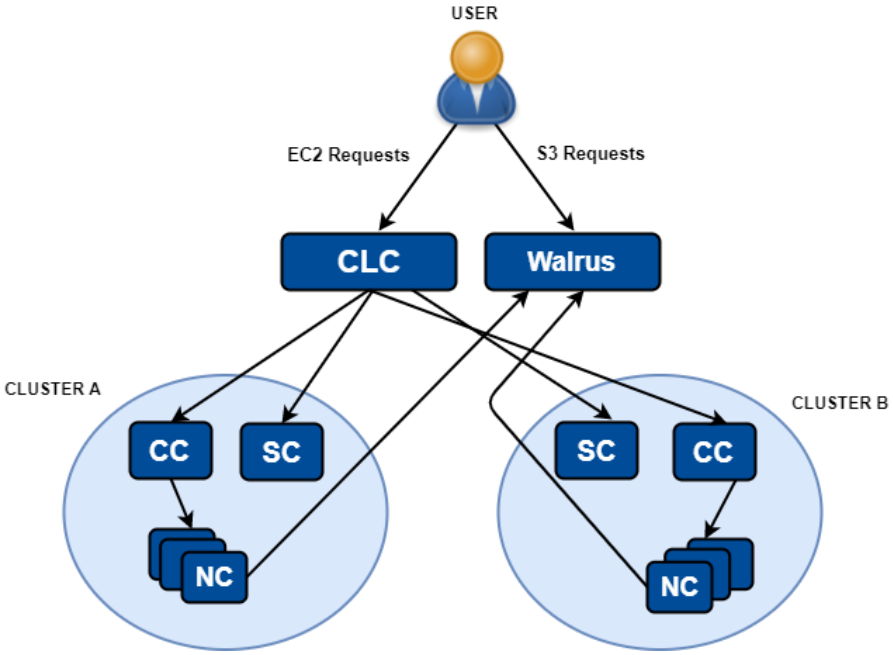

Figura 26 – Relacionamento entre componentes da nuvem Eucalyptus - Adaptado (JOHNSON et al., 2010b)

- ***Controlador de Nuvem* (CLC)**: Podemos chama-lo de *front-end* é responsável por expor e administrar os recursos subjacentes virtualizados (servidores, rede e armazenamento) via *API Amazon EC2*. Este componente utiliza interfaces de serviços da internet para receber os pedidos de ferramentas de clientes de um lado e de interagir com o resto dos componentes do Eucalyptus no outro lado. Com o CLC é possível monitorar as instâncias em execução e definir quais clusters ou agrupamentos deverão realizar o provisionamento de novas instâncias (DANTAS, 2013);

- ***Controlador de Cluster* (CC)**: Conhecido como *cluster controller* é o componente responsável por executar em um *cluster* de máquinas *front-end* ou em qualquer máquina que tenha conectividade na rede para ambos os nós que executam os controlador de nó e para a máquina que se encontra executando o controlador da nuvem. Os controladores de cluster reúnem informações sobre um conjunto de máquinas virtuais e horários de execução de máquinas virtuais no NCs específico (JOHNSON et al., 2010b).

- ***Controlador de Armazenamento* (SC)**: Ou *Storage Controller* é responsável por prover o armazenamento em blocos para as máquinas virtuais, sendo capaz de criar e gerenciar blocos de armazenamento e protegê-los de acesso indevido por outras máquinas em execução no nó (DANTAS, 2013) (JOHNSON et al., 2010a);

- ***Controlador de Nó* (NC)**: *Node Controller* é executado em cada nó físico e controla o ciclo de vida das instâncias em execução no nó. Ele interage com o sistema operacional e com o *hypervisor* em execução no nó. Os controladores de nós controlam a execução, fiscalização e terminação das instâncias de máquinas virtuais no *host* onde está sendo executado.

- **Walrus**: Responsável pelo armazenamento em arquivos, que permitem aos usuários o envio e coleta de informações e imagens de máquinas virtuais, através da Internet e seus protocolos possibilitam a migração e o *download* das máquinas virtuais.

6.1.2 Ambiente de Teste

O ambiente de teste contou uma nuvem privada composta por seis máquinas, duas com processador Core i5 de 3.20 GHz e 4GB RAM (Random Access Memory), duas com processador Core i3 de 2.93 GHz e duas com Core 2 de 2.66 GHz e 3GB RAM.

Em cinco máquinas foram instaladas servidores CentOS 5.4 (x86-64) que possuem a plataforma *Eucalyptus* na versão 3.2.2, onde temos quatro *nodes* e um *frontend*. A sexta máquina foi utilizada como cliente da nuvem e recebeu o sistema operacional Xubuntu 16.04.2 LTS (x86-64). A nuvem utilizada como teste é completamente baseada na plataforma *Eucalyptus* e utilizou-se do *hypervisor* KVM.

Portanto, instalou-se o *Cloud Controller (CLC)*, o *Cluster Controller (CC)*, o *Walrus* e o *Storage Controller (SC)* em um máquina física, ao qual chamamos de **Frontend**. Os Nodes ou Nó (Node Controller) foram instalados um em cada máquina física, visto que precisam gerenciar os recursos para suas respectivas VMs (Virtual Machine). Na máquina cliente foi instalado a ferramenta *Mercury* e o SIMF.

6.1.3 Arquitetura e Modelo

Na Figura 27 encontra-se a arquitetura básica do sistema na qual denominamos de *baseline*. Inicialmente dispomos: De uma máquina física que possui o *SIMF*, a *rede* responsável pela comunicação com a infraestrutura da nuvem, e a plataforma *Eucalyptus* que agrega: o *frontend* e dois *Nós*, cada *Nó* possui duas *VMs*, totalizando quatro. É importante ressaltar que esta arquitetura básica não possui serviços ativos, salvo os da própria nuvem. O principal objetivo deste estudo é agregar o modelo SPN desta *baseline* a um ambiente real utilizando o SIMF.

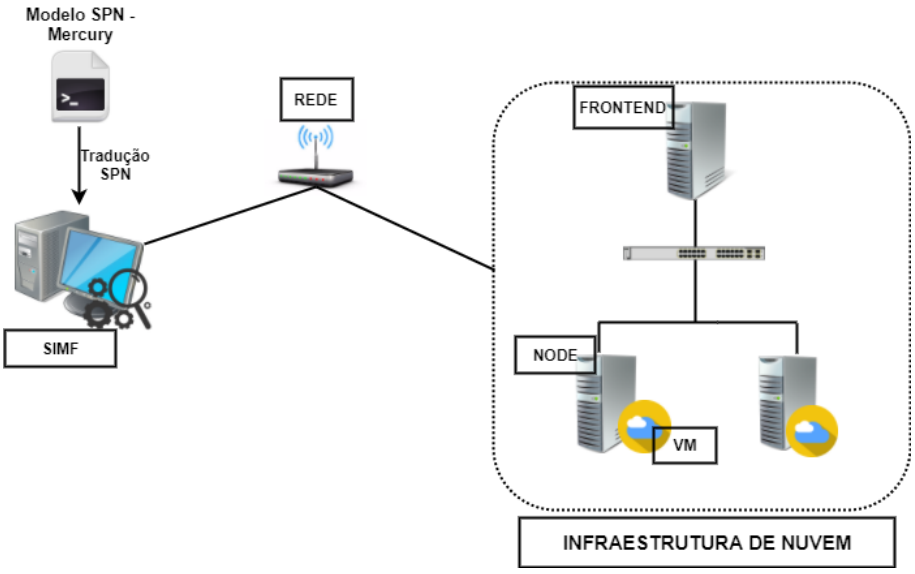

Figura 27 – Visão a nível de arquitetura

Apresentamos o modelo da SPN na Figura 28, o mesmo é baseado na arquitetura *baseline* referente a Figura 27. Este modelo é executado da seguinte forma: A *subred Frontend* é o controlador da nuvem (CLC) da infraestrutura *Eucalyptus*, responsável por administrar os recursos, quando o *token* encontra-se no lugar *frontOn*, entende-se que este componente está ativo. Quando a transição *failFront* é disparada, o *token* desloca-se do status On e torna-se inativo, ocupando o lugar *frontOff* denotando sua inatividade. Após sua inatividade, a transição *repairFront* é disparada movendo o *token* para o lugar inicial *frontOn* que refere-se a disponibilidade da rede.

No que se diz respeito a *subrede Node*, denominam-se aos nós físicos da infraestrutura que controlam o ciclo de vida das instâncias em execução. O *token* no lugar *nodeOn* e *nodeOn2* indica que o componente está ativo, quando a transição *failNode* e *failNode2* são disparada, o *token* é consumido e passa de ativo para inativo, ocupando o lugar *nodeOff* e *nodeOff2*. Quando os Nós da rede tornam-se inativos, o arco inibidor *TI1* e *TI2* são acionados e as máquinas virtuais (VM) instanciadas tornam-se inaptas automaticamente. As mesma só podem estar operacionais, caso os seus respectivos Nós estejam ativos.

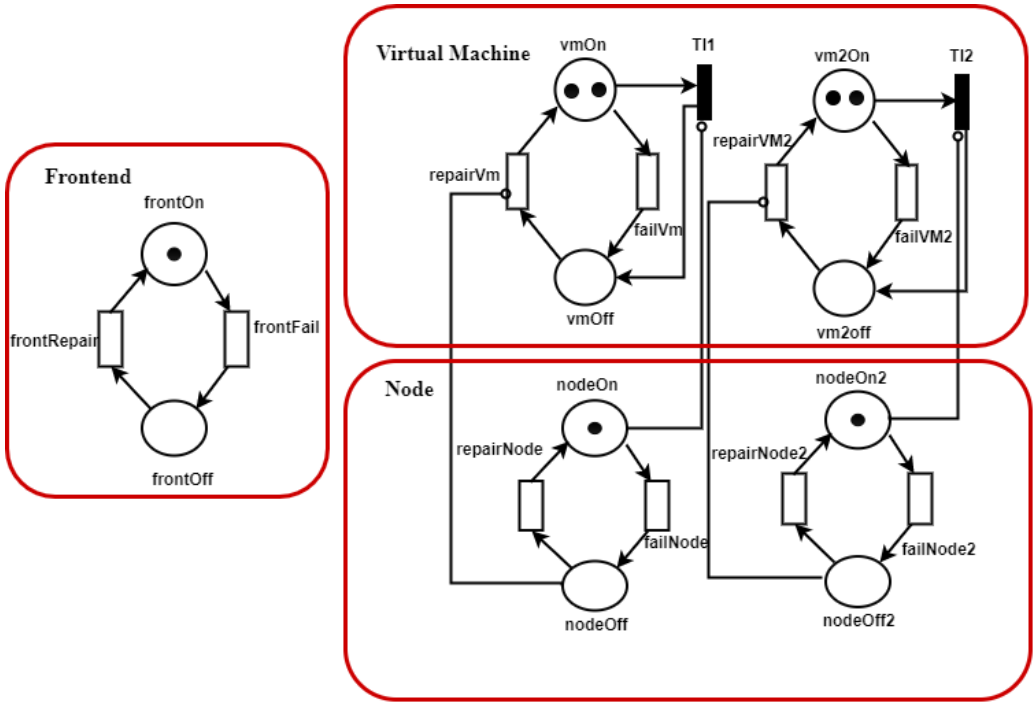

Figura 28 – Modelo SPN da arquitetura *baseline*

A *subrede Virtual machine* corresponde as VMs instanciadas pelos Nós da rede, existem dois *tokens* nos lugares *vmOn* e *vmOn2* indicam que as máquinas estão ativas. Quando a transição *failVM* e *failVM2* são disparadas os *tokens* são consumidos e ocupam os lugares *vmOff* e *vmOff2*. Após as condições serem satisfeitas, será habilitada a transição *repairVM* e *repairVM2* que representa o tempo de ativação das VMs.

Podemos observar o cálculo da disponibilidade deste modelo através da Equação:

$$DP\text{: P}\{((\#frontOn = 1))AND\,(((\#nodeOn = 1)$$
$$\text{AND } (\#vmOn > 0))$$
$$\text{OR } ((\#nodeOn2 = 1)$$
$$\text{AND}(\#vm2On > 0))))\}$$

6.1.4 Validação

Com o intuito de validar o modelo SPN da *baseline* em um cenário real, inserimos o referido na ferramenta SIMF, ao final do processo de injeção de falhas, aplicamos cálculos estatísticos propostos por (KEESEE, 1965) para verificar se a disponibilidade gerada pelo modelo SPN estava incluída no intervalo de confiança.

Em virtude da quantidade de tempo necessária para a ocorrência de falhas, os experimentos foram conduzidos levando em consideração valores de MTTF e MTTR reais, que são encontrados em: (DANTAS et al., 2015), (BEZERRA et al., 2014), (DANTAS, 2013) e (KIM; MACHIDA; TRIVEDI, 2009). Para a realização dos experimentos aplicou-se o fator de redução de 100 no *Frontend* e no *Node* , e de 1000 nas Máquinas Virtuais nos valores de MTTF reais. A aplicação do fator de redução propiciou executar experimentos mais rapidamente, em virtude de que se os valores reais fossem adotados, levariam semanas até que uma falha ocorresse e horas para que falhas fossem corrigidas o que demandaria muito tempo para execução de cada experimento, a injeção de falha ocorreu nos componentes do *frontende* CLC (Controlador da Nuvem) e CC (Controlador de Cluster), e no Nó (Controlador de Nó). O tempo de duração do experimento na ferramenta SIMF foi de **24 horas**, na Tabela 11, apresentamos os valores reduzido, que foram utilizados, tanto no sistema real quanto no modelo.

Tabela 11 – Valor dos componentes

Componente	**MTTF Real**	**MTTF Reduzido**	**MTTR**
CLC,CC e NC	788,4 h	7,884 h	1 h
Virtual Machine	2880 h	2,880 h	1 h

O objetivo principal desta validação é quantificar a confiança na capacidade de previsão do SIMF em ambientes experimentais. Além de demonstrar que o modelo SPN de fato, é uma representação do sistema real, e reproduz o comportamento do mesmo com fidelidade suficiente para satisfazer os objetivos da análise.

Logo após, a injeção de falhas utilizando o modelo SPN, realizamos a validação dos dados gerados pelo SIMF. O método de (KEESEE, 1965) consiste em calcular o intervalo de confiança da disponibilidade. Aplicamos o método da função de distribuição-F através da ferramenta *Minitab*, que considera o número de ocorrências de falhas e reparos no sistema e um intervalo de confiança.

O número total das ocorrências de falha e reparo, gerado por SIMF utilizando o modelo SPN, podemos observar na Tabela 12, cuja soma resulta no grau de liberdade da validação. O intervalo de confiança que consideramos é de 95%.

Tabela 12 – Total de Ocorrências

Descrição	**Valor**
Falhas	613
Reparos	613
Total	1226

A partir do grau de liberdade e do intervalo de confiança de 95% na função de distribuição- F, alcançamos o intervalo de mínimo e máximo da distribuição, como observamos na Tabela 13 :

Tabela 13 – Intervalo da Distribuição F

Descrição	**Valor**
Valor Mínimo	0,8940
Valor Máximo	1,119

De posse dos resultados de SIMF, através do monitor, que informa os dados obtivemos os tempos em que o serviço esteve disponível e indisponível no decorrer do experimento. A relação entre o tempo total de falhas e o tempo total de reparos, resulta no valor de ρ. Assim, podemos dizer que Y_n indica o tempo total reparos dos componentes dos serviços, e S_n indica o tempo total de falhas dos componentes dos serviços. Como mostra a Equação:

$$P = \frac{Y_n}{S_n} \tag{6.1}$$

Ao alcançar o valor de ρ, bem como dos valores de mínimo e máximo da distribuição da Tabela 13, podemos obter os valores de ρ_L e ρ_U. Esses valores são obtidos a partir da divisão de ρ pelos valores de mínimo e máximo da distribuição, alcançando respectivamente ρ mínimo (ρ_L) e ρ máximo (ρ_U) (KEESEE, 1965). Na Tabela 14 os valores de ρ alcançados.

Com os valores de mínimo e máximo de ρ alcançados, podemos obter o intervalo da disponibilidade. Esse intervalo é calculado a partir das Equações 6.2 e 6.3, que resultam na disponibilidade mínima (A_L) e máxima (A_U) da validação.

$$A_U = \frac{1}{1 + \rho_U} \tag{6.2}$$

A_L:

$$A_L = \frac{1}{1 + \rho_L} \tag{6.3}$$

Tabela 14 – Intervalo de confiança para A e ρ

Intervalo de Confiança de 95%		
ρ	ρ_U	0,220598888
	ρ_L	0,176242543
A	A_U	0,819269958
	A_L	0,850164794

Podemos verificar na Tabela 14, os valores de mínimo e máximo de ρ e de A. Desta forma, podemos confirmar através do cálculo da disponibilidade utilizando a técnica de (KEESEE, 1965), que o modelo SPN inserido dentro da ferramenta SIMF, encontra-se dentro do intervalo da disponibilidade, visto que a disponibilidade do sistema real (SIMF) encontra-se dentro dos limites das disponibilidades identificadas no modelo analítico antes da injeção, isto é, a ferramenta é confiável e estabelece uma relação positiva com modelo desenvolvido dentro da ferramenta Mercury, antes de ser introduzido ao SIMF. Podemos observar a disponibilidade na Tabela 15.

Tabela 15 – Disponibilidade do Modelo SPN Antes e Depois de utilizar SIMF

Disponibilidade do Sistema	**Valor**
Disponibilidade do Modelo	84,60%
Disponibilidade gerada por SIMF	83,53%

Na Tabela 16, podemos observar o valor da disponibilidade do modelo SPN: %0,8460; o segundo é a disponibilidade gerada por SIMF %0,8353 e ao final encontramos o intervalo de confiança. Observa-se que o valor da disponibilidade encontrado está dentro do intervalo de confiança o que nos indica que SIMF forneceu evidências que proporcionam inferir que o modelo da arquitetura base SPN, no qual foi incorporado ao mesmo é de fato válido sob o aspecto da disponibilidade da infraestrutura analisada. Dessa forma podemos determinar, que não existem evidencias que possam refutar este modelo.

Tabela 16 – Valor dos Modelos

Modelo	**SIMF**	**Intervalo**
0,8460	0,8353	[0,8192 ; 0,8501]

6.1.5 Análise de Sensibilidade

Aplicamos uma das estratégias de análise de sensibilidade ao ambiente de teste com o objetivo de identificar os componentes com maior relevância para a arquitetura. Para isso escolhemos a estratégia chamada de Diferença Percentual (DP). Essa estratégia calcula a análise de sensibilidade por meio da Equação 6.4. Esta equação apresenta a expressão

para essa abordagem, em que $max\{Y(\theta)\}$ e $min\{Y(\theta)\}$ são os valores máximo e mínimo de saída, respectivamente, calculados ao variar o parâmetro θ ao longo de uma gama de n possíveis valores de interesse. Se $Y(\theta)$ é conhecido para variar monotonicamente, de modo que apenas os valores extremos de θ (i.e.,θ_1 e θn) podem ser usados para calcular $max\{Y(\theta)\}$ e $min\{Y(\theta)\}$ e consequentemente $S_\theta\{Y\}$ matos2015sensitivity.

$$S_\theta\{Y\} = \frac{max\{Y(\theta)\} - min\{Y(\theta)\}}{max\{Y(\theta)\}} \tag{6.4}$$

Em que,

$$max\{Y(\theta)\} = max\{Y(\theta_1), Y(\theta_2), \cdots, Y(\theta_n)\} \tag{6.5}$$

e

$$min\{Y(\theta)\} = min\{Y(\theta_1), Y(\theta_2), \cdots, Y(\theta_n)\} \tag{6.6}$$

A Tabela 17 apresenta o *ranking* da análise de sensibilidade em ordem decrescente.

Tabela 17 – *Ranking* de sensibilidade por meio da Diferença Percentual

Parâmetro	$SS(\lvert A\rvert)$
$Front_\mu$	$0,112546862$
$Front_\lambda$	$0,092300747$
$VM2_\lambda$	$0,051527714$
$VM1_\lambda$	$0,051527701$
$N1_\mu$	$0,046727406$
$N2_\mu$	$0,046727406$
$N1_\mu$	$0,022854557$
$N2_\mu$	$0,022854557$
$VM2_\lambda$	$0,020805226$
$VM1_\lambda$	$0,020805225$

Essa estratégia destaca os componentes $Front_\mu$, $Front_\lambda$, $VM1_\lambda$ e $VM2\lambda$ são os componentes mais importantes para essa arquitetura.

Na estratégia Diferença Percentual, os valores de entrada adotados correspondem aos níveis máximos e mínimos de taxa de falha e de reparo do sistema. Foram adotados os níveis mínimos para as taxas de falha e reparo de acordo com (BEZERRA, 2015) (MATOS et al., 2015). No entanto, para os níveis máximos de taxa de falha, foi utilizado um período de falha de 4380 horas, 6 (seis) meses. Entendemos que esse prazo seja razoável para os componentes e subsistemas analisados, uma vez que não empregam a redundância. Por outro lado, os valores máximos adaptados as taxas de reparo foram em torno de 100% do valor mínimo, considerando que há uma equipe exclusiva de manutenção.

Tabela 18 – Parâmetros de entrada para a estratégia *DP*

Parâmetros	Níveis ($Horas$)	
-	Mínimo	Máximo
$Front_{\lambda}$	788,40	4380
$N1_{\lambda}$	788,40	4380
$N2_{\lambda}$	788,40	4380
$VM1_{\lambda}$	2880	4380
$VM2_{\lambda}$	2880	4380
$Front_{\mu}$	1	2
$N1_{\mu}$	1	2
$N2_{\mu}$	1	2
$VM1_{\mu}$	1	2
$VM2_{\mu}$	1	2

Para melhor entendimento da análise de sensibilidade dos componentes gerou-se dois gráficos, o primeiro que está na Figura 29 relaciona-se ao tempo de falha (MTTF) do *Frontend*, onde a linha azul é a disponibilidade dita como referência para o componente da nuvem, e a linha pontilhada refere-se a disponibilidade do sistema para os MTTF padrões do mesmo. Entende-se que este gráfico compreende a disponibilidade do *Frontend*, na medida em que aumentamos a taxa de falha do componente *Frontend*, observamos que a disponibilidade aumenta de forma significativa.

No segundo gráfico da Figura 30 observamos o MTTF (tempo de falha), do *Node* (Nó), a linha azul mostra o limiar da disponibilidade padrão para este componente e a linha tracejada, a disponibilidade do componente em um ambiente de injeção de falhas. Comparando com a Figura 29 que exibe o tempo de falha do *Frontend*, é observado que o componente de maior criticidade desta arquitetura é o *Frontend*.

Porém a infraestrutura de nuvem *Eucalyptus*, é integralmente alinhada, ou seja, um componente depende do outro, inserir um componente redundante do *Frontend* na infraestrutura, implicaria em adicionar serviços extras para compor a nuvem, devido essa problemática decidiu-se que a nova arquitetura, possuiria dois *Node* em sua infraestru-

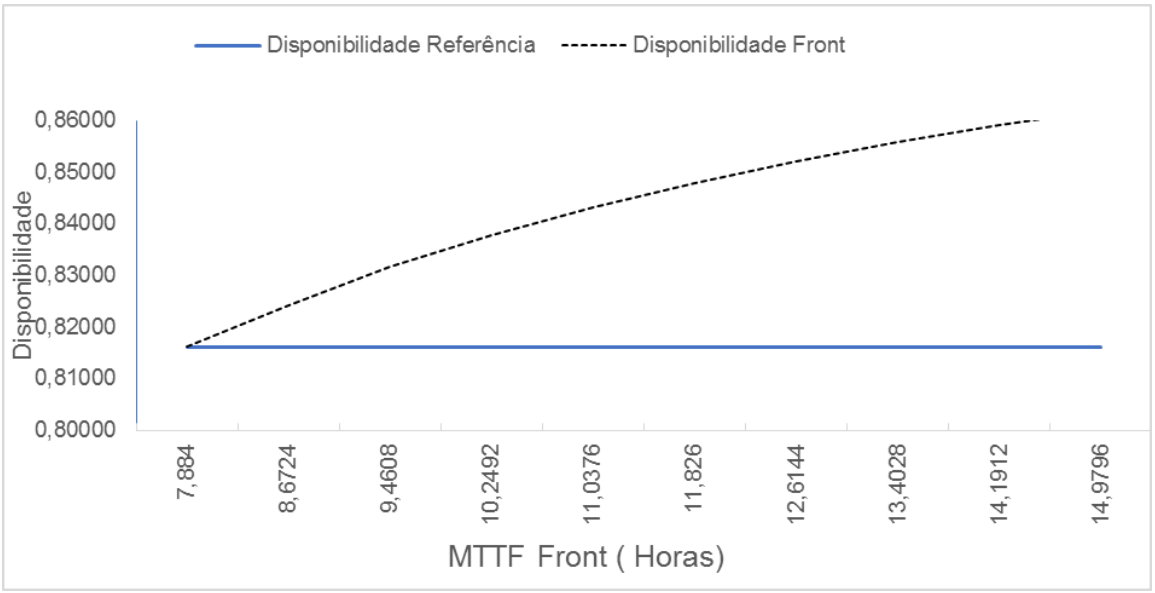

Figura 29 – Gráfico do tempo de falha do *Frontend*

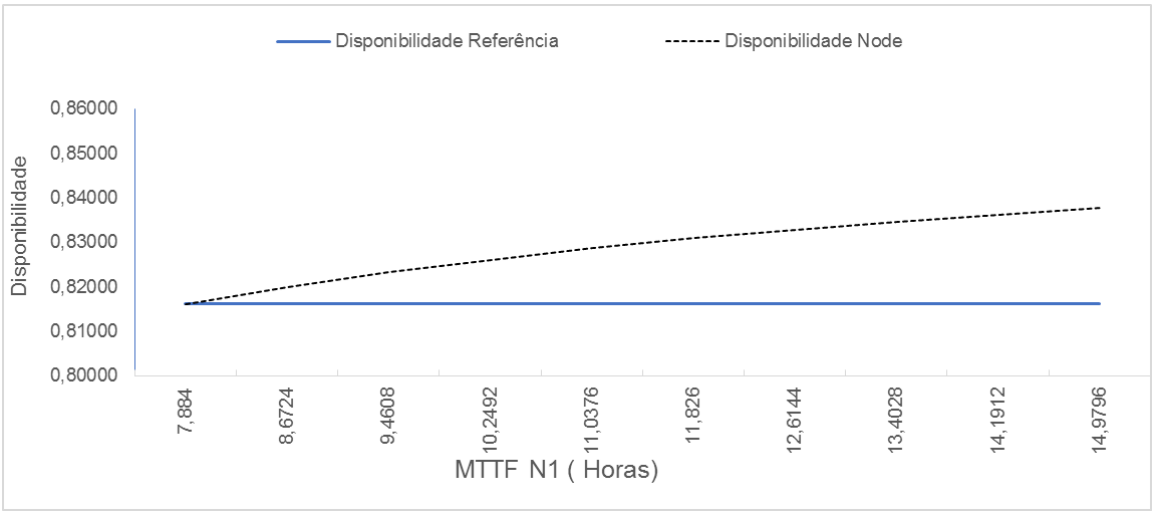

Figura 30 – Gráfico do tempo de falha do *Node*

tura. Visto que a VM era o segundo componente no *ranking* da análise de sensibilidade, implementou-se a redundância nos *Node* (Nó), o que consequentemente aumentou o numero de máquinas virtuais (VM). O modelo que possui *Node* a mais em sua infraestrutura, encontra-se no estudo de caso II.

6.2 Estudo de caso II

O segundo estudo de caso, foi decorrente da análise de sensibilidade utilizada no estudo de caso I, porém não foi levado em consideração o primeiro componente do *ranking* da análise, e sim o segundo que são as VMs, aumentou-se o número de Nodes na infraestrutura a fim de aumentar a capacidade. Neste segundo estudo de caso, primeiro

iremos abordar sobre o cenário utilizado, as máquinas da infraestrutura e o tempo de execução dos testes; Em seguida o modelo SPN e a arquitetura que originou o mesmo e finalizaremos com os resultados obtidos pela ferramenta.

6.2.1 Cenário do Ambiente

Baseado na análise de sensibilidade do estudo de caso I, aumentou-se os Nodes da infraestrutura *Eucalyptus*. A Figura 31, representa este novo sistema, a referida integra: um *Frontend*, quatro *Nós* cada um deles possuí duas *Vms*, totalizando, cinco maquinas físicas e oito virtuais. Não existem serviços na plataforma além dos oferecidos pela nuvem. Em decorrência do aumento da capacidade, foi possível calcular o número de máquinas virtuais criáveis por cada nó. Na Tabela 20, detalhamos o tipo de VM utilizada para esta arquitetura.

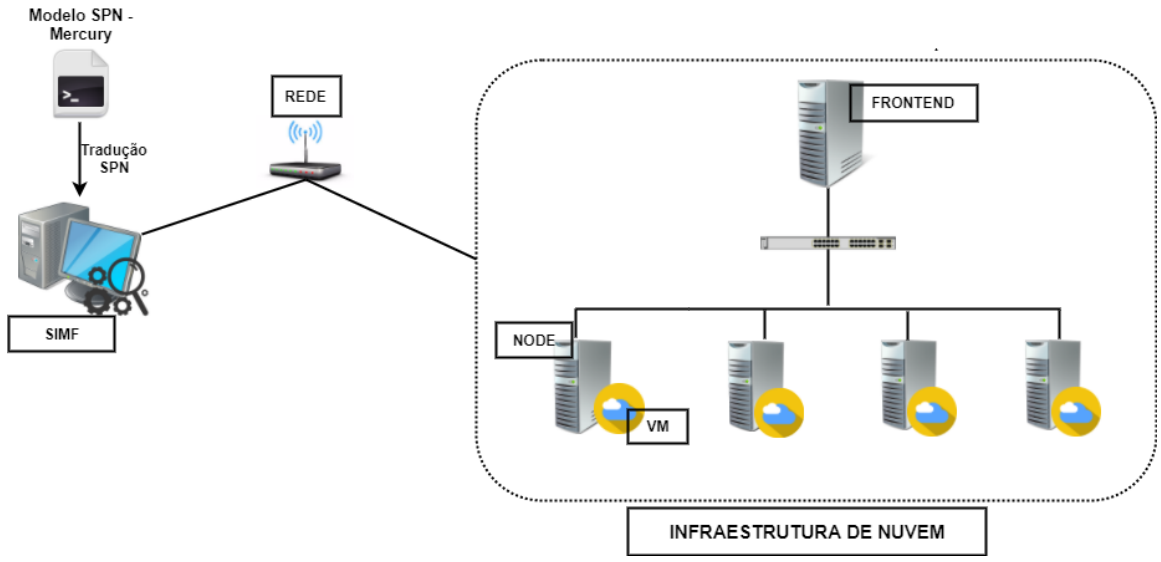

Figura 31 – Arquitetura do estudo de caso II

Foram adicionadas ao injetor **13** falhas referentes aos componentes da nuvem, *Frontend*, *Nó* e *VM*, para cada uma dessas falhas, foi adicionado reparo, isto é, contamos com **13** reparos. O experimento durou cerca de **48 horas** (2 dias), e contou com **2010** ocorrências de falha e reparo, sendo que foram **1005** para falha e o mesmo valor para reparo. Para a realização dos experimentos aplicou-se o fator de redução de 100 nos componentes CLC e CC do *Frontend* e NC do *Nó*, e de 1000 para as Máquinas Virtuais nos valores de MTTF. A aplicação do fator de redução permite que a execução dos experimentos ocorra de forma mais rápida, visto que levaria dias para que ocorresse uma falha em um componente. Os experimentos foram realizados levando em consideração valores de MTTF e MTTR reais obtidos em (DANTAS et al., 2015) (BEZERRA et al., 2014), (DANTAS, 2013) e (KIM; MACHIDA; TRIVEDI, 2009). Esses valores foram aplicados tanto no modelo SPN quanto na ferramenta SIMF, podemos observa-los na Tabela 19:

Tabela 19 – Valor dos componentes

Componente	**MTTF Real**	**MTTF Reduzido**	**MTTR**
CLC, CC e NC	788,4 h	7,884 h	1 h
Virtual Machine	2880 h	2,880 h	1 h

Tabela 20 – Tipos e Configurações de VM criáveis com Eucalyptus

Tipos de VM	CPU's Utilizáveis	Disco (GB)	RAM (MB)
m1.small	1	5	256
m1.medium	1	10	512
m1.large	2	10	512
m1.xlarge	2	10	1024

6.2.2 Modelo SPN Baseado na Arquitetura

É importante salientar que avaliou-se a disponibilidade do modelo SPN apresentado neste segundo estudo de caso, segundo a ferramenta de geração de eventos que o criou o Mercury, antes de validar o mesmo utilizando o SIMF. Os dados obtidos para que se calculasse esta disponibilidade foram retirados decorrente do processo de injeção de falha em SIMF.

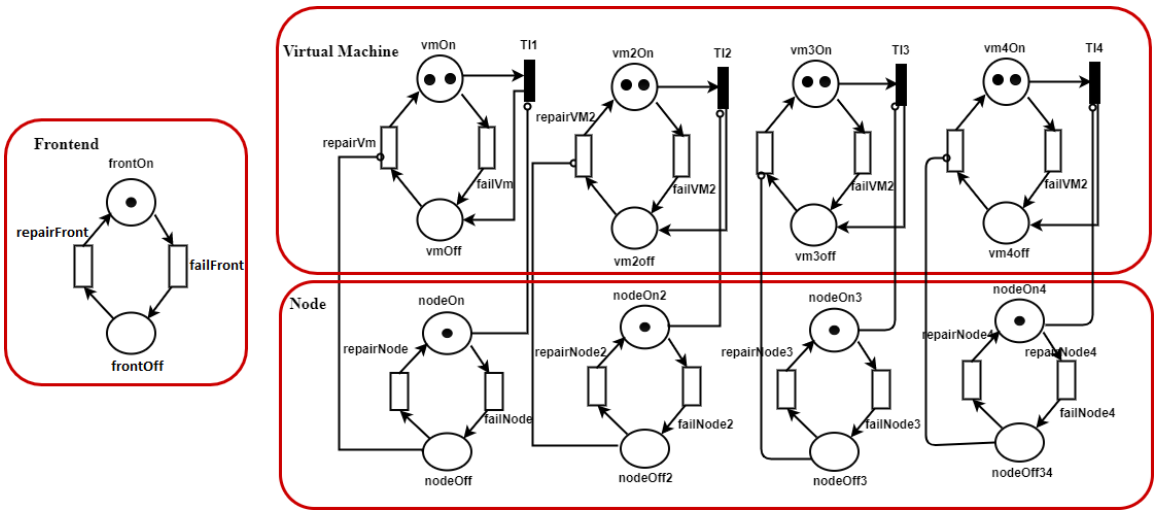

Figura 32 – Modelo SPN da arquitetura II

O modelo SPN da Figura 32, representa a arquitetura da Figura 31, onde: O *token* no lugar *frontOn* indica que este componente está ativo, quando a transição *failFront* é disparada, o *token* é consumido e transferido para o lugar *frontOff* implicando na inatividade deste componente. Ao ser disparada a transição *repairFront* implica em dizer que este componente foi reparado e que o referido voltou a estar disponível. Na subrede Node existem quatro nós físicos. O *token* nos lugares *nodeOn* ou (*nodeOn2*, *nodeOn3* e *nodeOn4*) indica que o nó está ativo, quando acontece uma falha no sistema a transição

failNode ou (*failNode2*, *failNode3* e *failNode4*) são disparadas movendo o *token* do lugar On para o lugar *nodeOff* e respectivamente nos outros nodes da rede, o que implica em dizer que o Nó está inativo. Quando os Nós são reparados é disparada a transição *repairNode* ou (*repairNode2, repairNode3 e repairNode4*), o *token* volta para o lugar On manifestando que este componente está disponível.

A subrede Virtual machine corresponde as VMs instanciadas pelo nó da rede, existem dois *tokens* nos lugares da SPN , indicando que cada nó possui duas VMs instanciadas . A condição para as VMs estarem funcionando corretamente são que os nós estejam operacionais. Caso algum *token* esteja nos lugares *nodeOff* e/ou (*nodeOff2*, *nodeOff3* e *nodeOff4*), a transição imediata dispara e transfere o *token* para *vmOff* e/ou (*vmOff2*, *vmOff3*, *vmOff4*).

Os arcos inibidores *T11*, *T12*,*T13* e *T14* denotam que no momento em que os Nós da rede tornam-se inoperantes, automaticamente as VMs estarão indisponíveis. E ocuparão o lugar disponível caso o nó seja restaurado ou, se os Nós da rede estiverem operantes e se a referida falhar será reparada.

6.3 Resultados

Utilizando o modelo SPN detalhado acima no *framework* SIMF, levou-se em consideração todas as propriedades estruturais do mesmo e ao final do processo de injeção de falhas foi possível calcular a disponibilidade. Levamos em consideração o valor da disponibilidade gerado pelo modelo antes de ser utilizado pelo *framework* (cenário real). O resultado encontrado por SIMF após a injeção de falhas foi de 95,59% e a disponibilidade do modelo antes da injeção foi de 96,27%, como pode-se observar na Tabela 21. Enquanto isso na Tabela 22, observa-se que o intervalo de confiança conclui que os valores do modelo antes e depois da injeção de falha, encontram-se dentro do intervalo de confiança. Para isso conclui-se que o modelo validado pela ferramenta neste estudo de caso, não possui evidências que venham refutá-lo.

Tabela 21 – Disponibilidade do Modelo SPN Antes e Depois de utilizar SIMF

Disponibilidade do Sistema	**Valor**
Disponibilidade do Modelo	96,23%
Disponibilidade gerada por SIMF	95,94%

Tabela 22 – Intervalo de Confiança - Modelos

Intervalo de Confiabilidade de 95%	**Valor**
Máximo	0,9627
Mínimo	0,9559

6.4 Considerações Finais

Este capítulo apresentou dois estudos de caso envolvendo a utilização do *framework* SIMF, que engloba o formalismo SPN como um mecanismo de injeção de falhas. Os cenários elaborados em ambos os estudos de caso procuraram representar situações envolvendo injeção de falha em ambientes de nuvem objetivando avaliar o impacto destas sobre a disponibilidade do sistema. Os resultados apresentados comprovam que a ferramentas atende os objetivos para qual foi construída. Ainda neste estudo, foi possível valer-se de uma técnica denominada análise de sensibilidade, através desta é possível observar o componente de maior criticidade do sistema.

No primeiro estudo de caso, com o auxilio da ferramenta SIMF injetamos falhas e reparos em um ambiente de teste real, com este resultado, encontramos a disponibilidade do modelo de 83,53%. Com base nos resultados foi aplicada a técnica de análise de sensibilidade, então foi possível encontrar os gargalos de disponibilidade deste cenário. A partir desta arquitetura, geramos um *ranking* de sensibilidade dos parâmetros da mesma, de posse desses valores, fomos capazes de construir um segundo estudo que caso. No segundo estudo de caso, decidimos aumentar a capacidade da nuvem, levando em consideração o componente Nó que estava em segundo lugar no *ranking*, utilizamos o SIMF para novamente validar o modelo SPN, e a ferramenta gerou uma disponibilidade de 95,94% . A escolha pelo aumento da capacidade, provou ser o melhor para o sistema.

Em virtude dos resultados gerados por SIMF, podemos concluir que esta ferramenta pode ser útil para pesquisadores que necessitam avaliar a disponibilidade de infraestrutura de nuvens computacionais, tal como desenvolver estratégias para melhoria da qualidade dos serviços suportados no ambiente, evitando o retrabalho por partes dos pesquisadores que encontram em SIMF a facilidade de analisar um modelo analítico SPN e validar o mesmo em um ambiente real.

7 CONSIDERAÇÕES FINAIS

Constantemente são desenvolvidas ferramentas para análise de sistemas entretanto as referidas tornam-se obsoletas, uma vez que em sua maioria, sejam especificas para um dado problema. Este trabalho propôs o desenvolvimento de um *framework* denominado SIMF com o intuito de diminuir o retrabalho por parte dos pesquisadores no que tange ao desenvolvimento de ferramentais limitados para a avaliação de disponibilidade em sistemas de nuvem computacional. O diferencial deste *framework* com relação a outras ferramentas avaliadas na literatura é beneficiar-se do formalismo matemático SPN e utilizar o mesmo como um mecanismo de injeção de falhas e reparo.

A ferramenta *Mercury* foi utilizada para a criação de modelos SPN, a partir desta foi possível gerar o script do modelo analisado e acoplá-lo a ferramenta SIMF. Decidimos incrementar na referida a parte que engloba simulação e *script* SPN. O *framework* SIMF, também fez uso de parte do ferramental de uma ferramenta denominada EucaBomber, no qual utilizou-se a conexão SSH2 e a geração de números aleatórios.

O diferencial de SIMF, foi incorporar essas duas realidades, a SPN e a injeção de falhas e reparo. O script do Mercury, é traduzido dentro de SIMF, onde o *places* é conciliado com o IP da rede. É importante salientar que SIMF leva em consideração todos os componentes da SPN. Além disso, foi observado que alguns tipos de ferramentas, não possuíam um monitor, que averiguasse a rede, e automaticamente imputasse estas informações ao usuário do que ocorria durante a fase de injeção de falhas em um sistemas, pensando nisso, desenvolvemos um monitor que exibe em tempo real, o funcionamento da nuvem durante o processo, além disso emite um relatório para análise dos dados obtidos.

Durante o processo de implementação das ferramentas observou-se que o tempo e o esforço empregado na codificação foi reduzido em parte, devido a redução na criação de classes e métodos necessários para desenvolver a ferramenta. Na etapa seguinte, foram realizados estudos de caso com intuito de demonstrar a eficiência de SIMF, nos dois estudos de caso, visou-se avaliar a disponibilidade de um ambiente em nuvem mediante a ocorrência de falhas, o diferencial deu-se no fato de que em ambos os estudos foram utilizados modelos SPN distintos, para a injeção na plataforma. No primeiro estudo de caso, utilizamos um modelo SPN mais simples, no qual foi avaliado o valor do modelo antes e depois de ser utilizado no SIMF. O segundo modelo era mais robusto, diferente do modelo I, foi adicionado dois componentes a infraestrutura, o Nó (*Node Controller*) e em cada Nó foi instanciado VMs, e então obtivemos a disponibilidade do mesmo com a ferramenta.

A eficiência de SIMF foi constatada através do cenário onde as falhas e reparos são injetados em uma nuvem de *Eucalyptus*. Os resultados dos testes foram comparados

aos resultados de um modelo SPN, antes de ser inserido na ferramenta, mostrando que o sistema sob a ação SIMF se comporta como esperado. Os cenários de teste também mostram que a ferramenta pode ajudar os administradores e planejadores do sistema a avaliar as políticas de disponibilidade e manutenção.

7.1 Contribuições

Este trabalho apresentou dois modelos analíticos que permitiram a avaliação de uma plataforma de computação em nuvem *Eucalyptus*. Para estimar métricas de disponibilidades, foi utilizado o formalismos matemático rede de Petri estocástica (SPN).

Visando analisar estes modelos em um ambiente de teste real, foi utilizado o *Framework* SIMF, que faz uso da SPN para injetar falha. A SPN foi escolhida por ser uma modelagem mais viável. Além disso, os modelos SPN são baseados em simulação.

Ao final, a ferramenta SIMF, constatou ser eficaz e eficiente nos resultados gerados. A mesma foi desenvolvida para ambientes de nuvem, e não unicamente para plataforma *Eucalyptus*, outras plataformas de nuvem podem utiliza-la. A mesma é adaptável, basta mudar os parâmetros utilizados na injeção de falhas e a comunicação. O *software* proposto atua inserindo e reparando falhas de acordo com tempos gerados baseados em distribuições exponencial.

SIMF é capaz de monitorar o ambiente de teste, com essa ferramenta é possível informar o status do componente em tempo real. A ferramenta irá auxiliar os estudantes da área de dependabilidade a não precisarem codificar ferramentas ou scripts, visto que podem validar seus sistemas utilizando o modelo SPN.

Como contribuição possuímos dois trabalho que estão aguardando resultados:

- *SIMF: Failure Monitoring and Injector Framework based on Petri Net for Evaluation of Availability on Cloud Computing* - Aguardando Resultado.
- *Cloud Availability Analysis Framework using SPN* - Em Conclusão.

7.2 Limitações e Trabalhos futuros

Nesta pesquisa foram observadas algumas limitações: Apesar de injetar falhas utilizando SPN a ferramenta não é capaz de traduzir um modelo, sem antes fazer uma chamada as transições dentro do código, não é algo que seja difícil, porém a mesma não possuí essa capacidade (tradução simultânea de modelo). A mesma situação acontece com os valores para injeção de falhas é preciso pré estabelece-los antes do teste. O SIMF possui uma funcionalidade para *hardware*, porém não foi possível fazer um estudo de *software* e *hardware* juntos pela limitação das máquinas. Houve dificuldade na codificação da ferramenta devido a união de duas ferramentas distintas, apesar de todos os subsídios e classes

tornar um formalismo matemático um mecanismo de injeção de falha não é uma tarefa trivial, foi preciso estudar conceitos e levar em consideração todos os componentes que pertencem a uma SPN, como arco de transição e etc.

Não foi possível utilizar outros serviços na plataforma de nuvem além dos existentes, devido ao tempo. Mas isso não quer dizer que a ferramenta, não possa avaliar outros modelos que utilizem serviços, SIMF é adaptável. Os testes foram realizados somente na plataforma *Eucalyptus* mas é possível fazer uso da mesma em outras plataformas de nuvem.

A ferramenta *Eucalyptus* apresentava problemas quando uma grande quantidade de falhas era injetada e reparadas em um curto espaço de tempo. Por muitas vezes os testes eram interrompidos e reiniciados devido ao mal comportamento da nuvem.

Durante o desenvolvimento do trabalho identificamos alguns pontos que deveriam ser classificados como extensões do trabalho atual para atividades futuras. Dentre os quais, podemos destacar: i) A tradução simultânea do *scipt* SPN, em vez de executar as chamadas dentro do código a tradução poderia ocorrer de forma instantânea; ii) utilizar plataformas de nuvens distintas; iii) apresentar um estudo comparativo com relação as disponibilidades geradas; iv) utilização de outros serviços na inserção de falhas utilizando SPN; v) junção de um modelo SPN de *hardware* e *software* com máquinas mais robustas para calcular a disponibilidade de um sistema como um todo.

Por fim, através da análise de sensibilidade encontramos outros modelos a serem analisados, como uma redundância no qual agregaria um *frontend* na infraestrutura do estudo de caso I, e poderia ser expandido para o estudo de caso II.

REFERÊNCIAS

ARLAT, J.; AGUERA, M.; AMAT, L.; CROUZET, Y.; FABRE, J. C.; LAPRIE, J. C.; MARTINS, E.; POWELL, D. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, v. 16, n. 2, p. 166–182, 1990. ISSN 00985589. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=44380>.

AVIZIENIS, a.; LAPRIE, J. C.; RANDELL, B. Fundamental Concepts of Dependability. *Technical Report Seriesuniversity of Newcastle Upon Tyne Computing Science*, v. 1145, p. 7–12, 2001. ISSN 13681060.

BALBO, G. Stochastic Petri net simulation. *of the 21st conference on Winter simulation*, p. 266–276, 1989. Disponível em: <http://www.osti.gov/energycitations/product.biblio.jsp?osti{_}id=6686401{\%}5Cnhttp://dl.acm.org/citation.cfm?id>.

BALBO, G. Introduction to stochastic Petri nets. *Lectures on Formal Methods and PerformanceAnalysis*, p. 84–155, 2001.

BANZAI, T.; KOIZUMI, H.; KANBAYASHI, R.; IMADA, T.; HANAWA, T.; SATO, M. D-cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology. In: IEEE. *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. [S.l.], 2010. p. 631–636.

BARAZA, J.-C.; GRACIA, J.; GIL, D.; GIL, P. A prototype of a VHDL-based fault injection tool. *Defect and Fault Tolerance in VLSI Systems, 2000. Proceedings. IEEE International Symposium on*, v. 47, p. 396–404, 2000. ISSN 1550-5774.

BAUER, E.; ADAMS, R.; EUSTACE, D. *Beyond redundancy: how geographic redundancy can improve service availability and reliability of computer-based systems*. [S.l.]: John Wiley & Sons, 2011.

BEZERRA, M. C.; MELO, R.; DANTAS, J.; MACIEL, P.; VIEIRA, F. Availability modeling and analysis of a vod service for eucalyptus platform. In: IEEE. *Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference on*. [S.l.], 2014. p. 3779–3784.

BEZERRA, M. C. d. S. Modelos para análise de disponibilidade de arquitetura de um serviço de vod streaming na nuvem. Universidade Federal de Pernambuco, 2015.

BOLCH, G.; GREINER, S.; MEER, H. de; TRIVEDI, K. S. Queueing Networks and Markov Chains. 1998. Disponível em: <http://doi.wiley.com/10.1002/0471200581>.

CANEDO, E. D. Modelo de confianca para a troca de arquivos em uma nuvem privada. 2013.

COSTA, I. d. O. Modelos par análise de disponibilidade em uma plataforma de mobile backend as a service. Universidade Federal de Pernambuco, 2015.

COSTA, I. d. O. *MODELOS PARA ANÁLISE DE DISPONIBILIDADE EM UMA PLATAFORMA DE MOBILE BACKEND AS A SERVICE*. Tese (Doutorado), 2015.

DANTAS, J.; MATOS, R.; ARAUJO, J.; MACIEL, P. Eucalyptus-based private clouds: availability modeling and comparison to the cost of a public cloud. *Computing*, v. 97, n. 11, p. 1121–1140, 2015.

DANTAS, J. R. Modelos para Análise de Dependabilidade de Arquiteturas de Computação em Nuvem. 2013.

EUCALYPTUS SYSTEMS, INC. *Eucalyptus Open-Source Cloud Computing Infrastructure - An Overview*. 130 Castilian Drive, Goleta, CA 93117 USA, 2009.

FAYAD, M.; SCHMIDT, D. C.; JOHNSON, R. E. *Building application frameworks: object-oriented foundations of framework design*. [S.l.: s.n.], 1999.

FUJITA, H.; MATSUNO, Y.; HANAWA, T.; SATO, M.; KATO, S.; ISHIKAWA, Y. DS-Bench Toolset: Tools for Dependability Benchmarking with Simulation and Assurance. *Proc. of the DSN - Intl. Conf. on Dependable Systems and Networks*, p. 1–8, 2012. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6263915>.

GALINDO, H. E. S.; SANTOS, W. M.; MACIEL, P. R. M.; SILVA, B.; GALDINO, M. L. Synthetic workload generation for capacity planning of virtual server environments. n. October, p. 2837–2842, 2009.

GERMAN, R. *Performance analysis of communication systems with non-Markovian stochastic Petri nets*. [S.l.]: John Wiley & Sons, Inc., 2000.

GUEDES, G. T. *UML: uma abordagem prática*. [S.l.]: Novatec Editora, 2008.

GUIMARÃES, A. P.; MACIEL, P. R.; MATIAS, R. An analytical modeling framework to evaluate converged networks through business-oriented metrics. *Reliability Engineering & System Safety*, Elsevier, v. 118, p. 81–92, 2013.

HSUEH, M.-C.; TSAI, T. K.; IYER, R. K. Fault injection techniques and tools. *Computer*, IEEE, v. 30, n. 4, p. 75–82, 1997.

JOHNSON; MURARI, K.; RAJU, M.; RB, S.; GIRIKUMAR, Y. *Eucalyptus Beginner's Guide*. Uec. [S.l.], 2010. For Ubuntu Server 10.04 - Lucid Lynx, v1.0.

JOHNSON, D.; MURARI, K.; RAJU, M.; SUSEENDRAN, R.; GIRIKUMAR, Y. Eucalyptus beginner's guide-uec edition. *Ubuntu Server*, 2010.

JOHNSON, R. E.; FOOTE, B. Designing Reuseable Classes. *Journal of object-oriented programming*, v. 2, n. 1, p. 22–35, 1988.

KEESEE, W. *A Method of Determining a Confidence Interval for Availability*. [S.l.], 1965.

KIM, D. S.; MACHIDA, F.; TRIVEDI, K. S. Availability modeling and analysis of a virtualized system. In: IEEE. *Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on*. [S.l.], 2009. p. 365–371.

LAJOIE, R.; KELLER, R. K. Design and reuse in object-oriented frameworks: Patterns, contracts, and motifs in concert. *Object-Oriented Technology for Database and Software Systems*, Singapore: World Scientific Publishing, p. 295–312, 1995.

MACIEL, P.; TRIVEDI, K.; MATIAS, R.; KIM, D. Performance and dependability in service computing: Concepts, techniques and research directions, ser. *Premier Reference Source. Igi Global*, 2011.

MACIEL, P. R. M.; MATIAS, R.; DONG, J.; KIM, S. Dependability Modeling. *Techniques*, n. 3, p. 53–55, 2012.

MACIEL, Paulo R M; LINS, Rafael D;CUNHA, P. R. M. *Introdução a Rede de Petri e Aplicações*. [S.l.: s.n.], 1996. v. 1. 201 p. ISSN 1098-6596. ISBN 9788578110796.

MARSAN, M. A.; CONTE, G.; BALBO, G. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 2, n. 2, p. 93–122, 1984.

MARSAN, M. A. P. d. T.; BALBO, G. U. d. T.; CONTE, G. U. d. P.; DONATELLI, S. U. d. T.; FRANCESCHINIS, G. U. d. P. O. Modelling with generalised stochastic petri nets. *System*, p. 299, 1994.

MARWAH, M.; MACIEL, P.; SHAH, A.; SHARMA, R.; CHRISTIAN, T.; ALMEIDA, V.; ARAÚJO, C.; SOUZA, E.; CALLOU, G.; SILVA, B. et al. Quantifying the sustainability impact of data center availability. *ACM SIGMETRICS Performance Evaluation Review*, ACM, v. 37, n. 4, p. 64–68, 2010.

MATOS, R.; ARAUJO, J.; OLIVEIRA, D.; MACIEL, P.; TRIVEDI, K. Sensitivity analysis of a hierarchical model of mobile cloud computing. *Simulation Modelling Practice and Theory*, Elsevier, v. 50, p. 151–164, 2015.

MATTSON, M. Object-Oriented Frameworks–A survey of methodological issues. 1996, 130p. *Tese (Licenciatura)–Department of ...*, p. 96–167, 1996. Disponível em: <http://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle: Object-Oriented+Frameworks+A+survey+of+methodological+is>.

MELL, P.; GRANCE, T. et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 2011.

OLIVEIRA, D. M. Análise de Disponibilidade e Consumo Energético em Ambientes de Mobile Cloud Computing. 2014.

OLIVEIRA, D. M. Análise de disponibilidade e consumo energético em ambientes de mobile cloud computing. Universidade Federal de Pernambuco, 2014.

SILVA, B. A framework for availability, performance and survivability evaluation of disaster tolerant cloud computing systems. Universidade Federal de Pernambuco, 2016.

SILVA, B.; CALLOU, G.; TAVARES, E.; MACIEL, P.; FIGUEIREDO, J.; SOUSA, E.; ARAUJO, C.; MAGNANI, F.; NEVES, F. ASTRO: An integrated environment for dependability and sustainability evaluation. *Sustainable Computing: Informatics and Systems*, Elsevier Inc., v. 3, n. 1, p. 1–17, 2013. ISSN 22105379. Disponível em: <http://dx.doi.org/10.1016/j.suscom.2012.10.004>.

SILVA, B.; MATOS, R.; CALLOU, G.; FIGUEIREDO, J.; OLIVEIRA, D.; FERREIRA, J.; DANTAS, J.; LOBO, A.; ALVES, V.; MACIEL, P. Mercury: An integrated environment for performance and dependability evaluation of general systems. In: *Proceedings of Industrial Track at 45th Dependable Systems and Networks Conference, DSN*. [S.l.: s.n.], 2015.

SILVA, F. A.; RODRIGUES, M.; MACIEL, P.; KOSTA, S.; MEI, A. Planning Mobile Cloud Infrastructures Using Stochastic Petri Nets and Graphic Processing Units. *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, p. 471–474, 2015. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7396196>.

SOUSA, E. T. G. d. Modelagem de desempenho, dependabilidade e custo para o planejamento de infraestruturas de nuvens privadas. UNIVERSIDADE FEDERAL DE PERNAMBUCO, 2015.

SOUSA, F. R.; MOREIRA, L. O.; MACHADO, J. C. Computacao em nuvem autonoma: Oportunidades e desafios. 2011.

SOUZA, D. S. L. d. Flexloadgenerator-um framework para apoiar o desenvolvimento de ferramentas voltadas a estudos de avaliação de desempenho e dependabilidade. 2013.

TRIVEDI, K. S. *Probability & statistics with reliability, queuing and computer science applications*. [S.l.]: John Wiley & Sons, 2008.

VAQUERO, L. M.; RODERO-MERINO, L.; CACERES, J.; LINDNER, M. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, ACM, v. 39, n. 1, p. 50–55, 2008.

VELTE, A. T.; VELTE, T. J.; ELSENPETER, R. C.; ELSENPETER, R. C. *Cloud computing: a practical approach*. [S.l.]: McGraw-Hill New York, 2010.

WEBER, T. S. Tolerancia a falhas: conceitos e exemplos. *Apostila do Programa de Pós-Graduação–Instituto de Informática-UFRGS. Porto Alegre*, 2003.

ZIADE, H.; AYOUBI, R. A.; VELAZCO, R. et al. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, v. 1, n. 2, p. 171–186, 2004.

APÊNDICE A – **MERCURY**

A ferramenta Mercury (SILVA et al., 2015a) foi gerada através do kernel de outra ferramenta denominada ASTRO (SILVA et al., 2013). A ferramenta ASTRO foi criada para a avaliação de sustentabilidade, custo e disponibilidade de ambientes de data center, a mesma permite que usuários modelem sistemas de TI, sistemas energéticos e sistemas de resfriamento, sem a necessidade de conhecer os formalismos matemáticos que serão utilizados para realizar a análise das propriedades desses sistemas (OLIVEIRA, 2014).

Por sua vez, Mercury (SILVA et al., 2015a) é utilizada para a criação e análise de modelos de RBD, cadeias de Markov, redes de Petri estocásticas (SPN) e modelos de fluxo energético (EFM). Esse ambiente foi desenvolvido pelo grupo modcs. A referida oferece outros recursos como análise de Sensibilidade dos modelos CTMC , RBD e simulação SPN. Para auxiliar o usuário na avaliação de modelos SPN, o Mercury possui um jogo simbólico que é chamado de *tokengame*, que torna possível o disparo das transições graficamente, os disparos iniciam-se de acordo com a marcação atual da Rede de Petri e assim os usuários podem avaliar as propriedades estruturais da SPN.

A aproximação de uma distribuição empírica por uma distribuição *phase-type* (hipoexponencial, hiperexponencial ou Erlang) através da técnica de *moment matching* . Cadeias de Markov e redes de Petri estocásticas exigem que os tempos de transição entre os estados sigam uma distribuição exponencial. Quando desejamos parametrizar o tempo de uma transição através de dados empíricos que se desviam da natureza exponencial, uma solução é aproximar estes dados por uma distribuição *phase-type*. Tais distribuições são formadas pela composição de fases exponenciais e desta forma, possibilitam a sua utilização em modelos markovianos (OLIVEIRA, 2014).

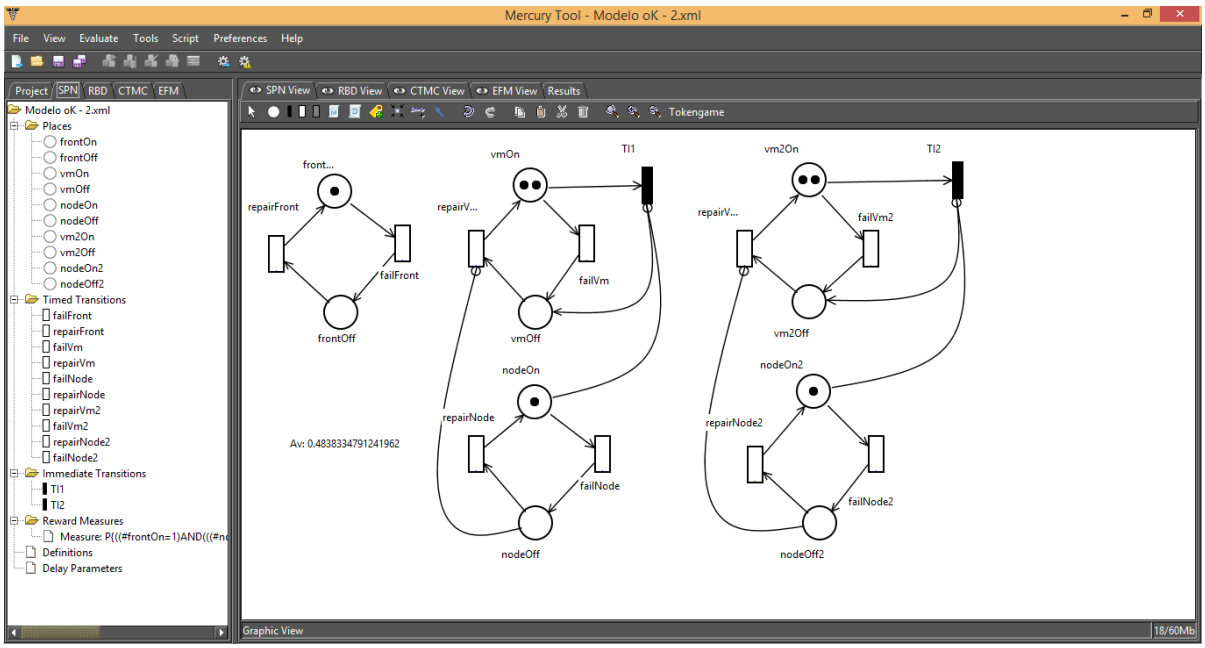

Figura 33 – *Screenshot* Mercury

APÊNDICE B – TUTORIAL DA FERRAMENTA SIMF

A ferramenta SIMF possuí interfaces gráfica que proporcionam ao usuário facilidade com a comunicação do sistema. Na Figura 34 o primeiro dado a ser informado corresponde ao arquivo de script SPN gerado pelo Mercury, após o carregamento do arquivo o injetor fará uma leitura automática de todas as transições da rede e carregará uma tela, solicitando informações adicionais informações (IP do servidor, senha e tempo de duração do teste) observe a Figura 35, para iniciar o processo de injeção na rede, o preenchimento desses campos é estritamente necessário, visto que o protocolo SSH2 necessita dessas informações para estabelecer a comunicação com o servidor *Eucalyptus*.

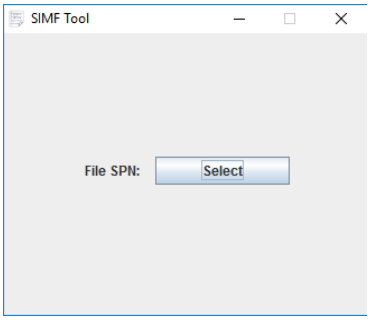

Figura 34 – *Screenshot* Interface Inicial

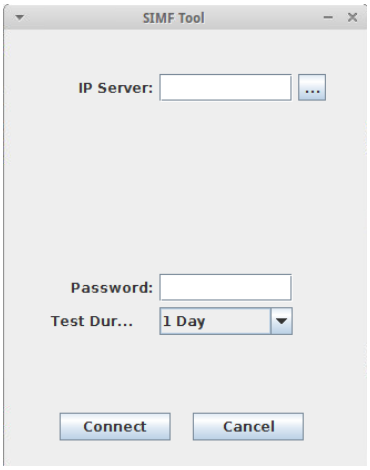

Figura 35 – *Screenshot* Informações da Rede

Na Figura 36 é necessário que o usuário digite o nome da máquina, coloque uma virgula e o IP que representa este componente físico, por exemplo: *node, 192.168.0.67*.

Ao lado do campo em branco do IP *server*, temos uma "caixinha"para acrescentar, outros IPs a rede, caso a SPN, seja maior do que os campos informados. O menu duração

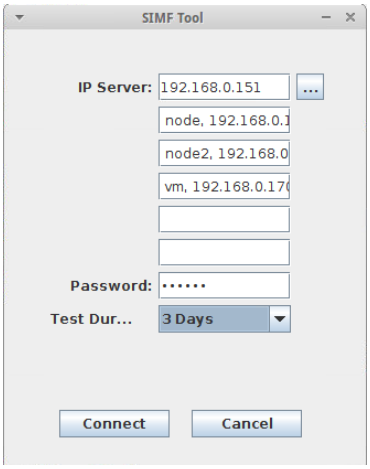

Figura 36 – *Screenshot* Informações da Rede II

do teste foi utilizado com o objetivo es especificar a duração do teste. Evitando que o usuário pare o teste manualmente.

A interface gráfica, apresentada na Figura 37, retrata a mensagem informando que a conexão com a nuvem foi estabelecida e está aguardando a confirmação do usuário com o clique no botão "OK" para iniciar-se o processo, após o usuário clicar no botão "OK" da mensagem não será mais necessária nenhuma intervenção, os passos seguintes serão realizados automaticamente até o final do processo de injeção.

Caso o usuário clique no botão *Connect* e a mensagem de confirmação da conexão não seja exibida, significa que houve algum problema com a comunicação entre o SIMF e o servidor, portanto é importante que o usuário deixe o ambiente, somente após a mensagem ser exibida e o botão OK ser pressionado. Abaixo observa-se o monitor da rede, no qual informa ao usuário o status da mesma, tal como a quantidade de falhas e reparos, o momento que esteve ativa ou inativa e no final entrega um relatório com os dados da injeção.

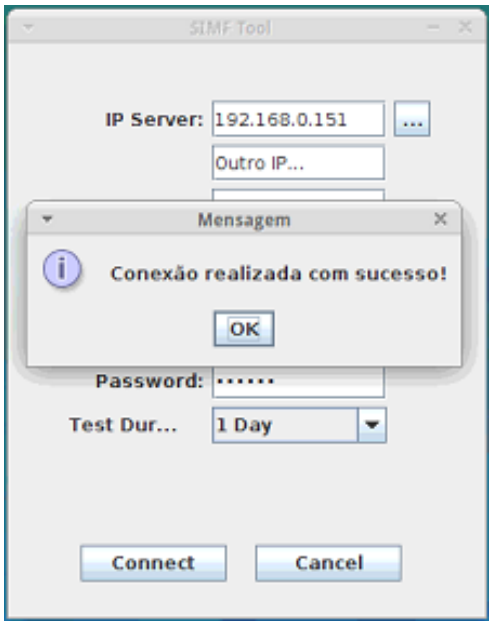

Figura 37 – *Screenshot* Informações do usuário

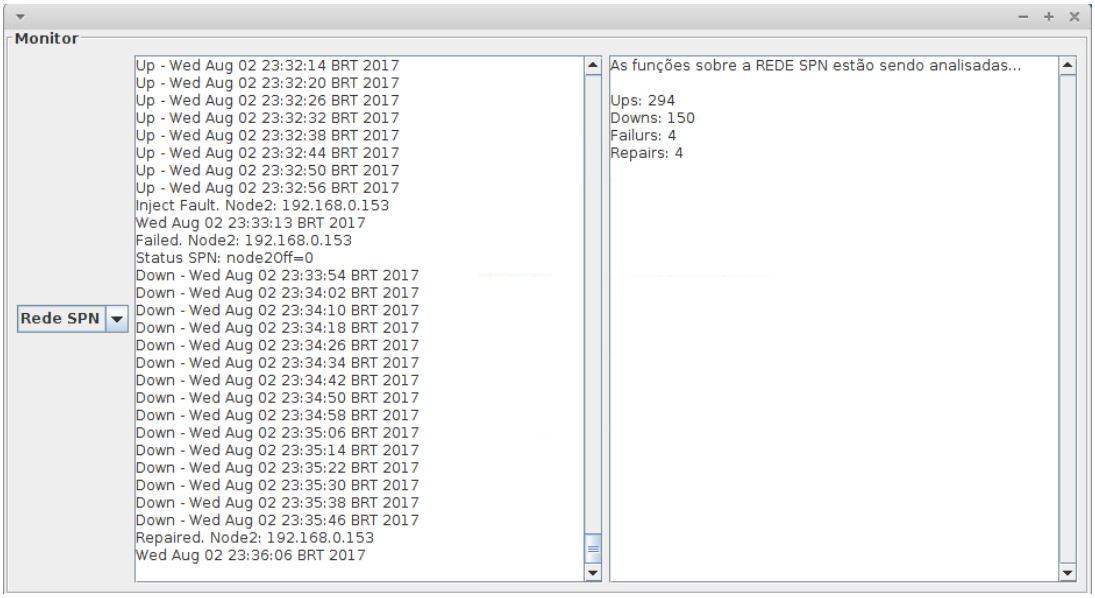

Figura 38 – *Screenshot* Monitor

APÊNDICE C – **DIAGRAMA DA FERRAMENTA EUCABOMBER**

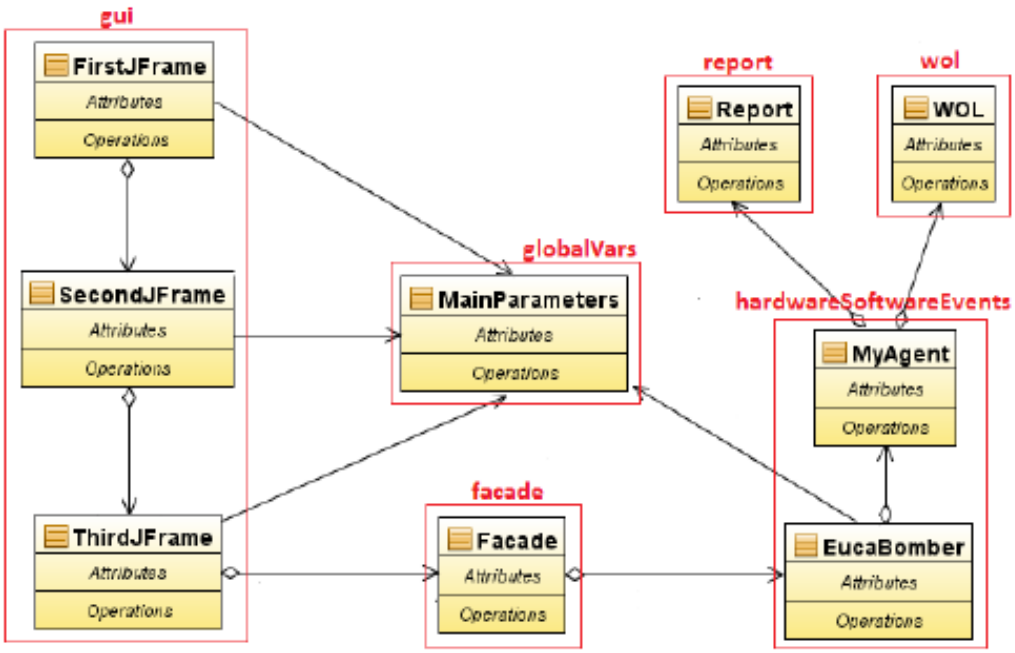

Figura 39 – *Screenshot* Diagrama *EucaBomber*

APÊNDICE D – SCRIPT DO MERCURY - ESTUDO DE CASO I

Listing D.1 – Script SPN do Mercury para tradução no SIMF

```
SPN Model{

place frontOff;
place frontOn( tokens= 1 );
place nodeOff;
place nodeOff2;
place nodeOn( tokens= 1 );
place nodeOn2( tokens= 1 );
place vm2Off;
place vm2On( tokens= 2 );
place vmOff;
place vmOn( tokens= 2 );

immediateTransition TI1(
inputs = [vmOn],
outputs = [vmOff],
inhibitors = [nodeOn]
);

immediateTransition TI2(
inputs = [vm2On],
outputs = [vm2Off],
inhibitors = [nodeOn2]
);

timedTransition failFront(
inputs = [frontOn],
outputs = [frontOff],
delay = 7.884
);

timedTransition failNode(
inputs = [nodeOn],
outputs = [nodeOff],
delay = 7.884
);

timedTransition failNode2(
inputs = [nodeOn2],
outputs = [nodeOff2],
delay = 7.884
);

timedTransition failVm(
inputs = [vmOn],
outputs = [vmOff],
```

```
delay = 2.88
);

timedTransition failVm2(
inputs = [vm2On],
outputs = [vm2Off],
delay = 2.88
);

timedTransition repairFront(
inputs = [frontOff],
outputs = [frontOn],
delay = 1.0
);

timedTransition repairNode(
inputs = [nodeOff],
outputs = [nodeOn],
delay = 1.0
);

timedTransition repairNode2(
inputs = [nodeOff2],
outputs = [nodeOn2],
delay = 1.0
);

timedTransition repairVm(
inputs = [vmOff],
outputs = [vmOn],
inhibitors = [nodeOff],
delay = 1.0
);

timedTransition repairVm2(
inputs = [vm2Off],
outputs = [vm2On],
inhibitors = [nodeOff2],
delay = 1.0
);

metric Av = stationaryAnalysis( expression = "P{((#frontOn=1)AND(((#nodeOn=1)AND(#
    vmOn>0))OR((#nodeOn2=1)AND(#vm2On>0))))}" );
}

main {
Av = solve( Model,Av );
println(Av);

}
}
```

APÊNDICE E – SCRIPT DO MERCURY - ESTUDO DE CASO I

Listing E.1 – Script SPN do Mercury para tradução no SIMF

```
SPN Model{

place Vm4Off;
place frontOff;
place frontOn( tokens= 1 );
place node2Off;
place node2On( tokens= 1 );
place node3On( tokens= 1 );
place node4Off;
place node4On( tokens= 1 );
place nodeOff;
place nodeOff3;
place nodeOn( tokens= 1 );
place vm2Off;
place vm2On( tokens= 2 );
place vm3Off;
place vm3On( tokens= 2 );
place vm4On( tokens= 2 );
place vmOff;
place vmOn( tokens= 2 );


immediateTransition TI0(
inputs = [vm4On],
outputs = [Vm4Off],
inhibitors = [node4On]
);

immediateTransition TI1(
inputs = [vmOn],
outputs = [vmOff],
inhibitors = [nodeOn]
);

immediateTransition TI2(
inputs = [vm2On],
outputs = [vm2Off],
inhibitors = [node2On]
);

immediateTransition TI3(
inputs = [vm3On],
outputs = [vm3Off],
inhibitors = [node3On]
);

timedTransition TE0(
```

```
    inputs = [frontOn],
49  outputs = [frontOff],
    delay = 7.884
51  );

53  timedTransition TE1(
    inputs = [frontOff],
55  outputs = [frontOn],
    delay = 1.0
57  );

59  timedTransition failNode(
    inputs = [nodeOn],
61  outputs = [nodeOff],
    delay = 7.884
63  );

65  timedTransition failNode2(
    inputs = [node2On],
67  outputs = [node2Off],
    delay = 7.884
69  );

71  timedTransition failNode3(
    inputs = [node3On],
73  outputs = [nodeOff3],
    delay = 7.884
75  );

77  timedTransition failNode4(
    inputs = [node4On],
79  outputs = [node4Off],
    delay = 7.884
81  );

83  timedTransition failVm(
    inputs = [vmOn],
85  outputs = [vmOff],
    delay = 2.88
87  );

89  timedTransition failVm2(
    inputs = [vm2On],
91  outputs = [vm2Off],
    delay = 2.88
93  );

95  timedTransition failVm3(
    inputs = [vm3On],
97  outputs = [vm3Off],
    delay = 2.88
99  );

101 timedTransition failVm4(
    inputs = [vm4On],
103 outputs = [Vm4Off],
    delay = 2.88
```

```
105 );

107 timedTransition repairNode(
    inputs = [nodeOff],
109 outputs = [nodeOn],
    delay = 1.0
111 );

113 timedTransition repairNode2(
    inputs = [node2Off],
115 outputs = [node2On],
    delay = 1.0
117 );

119 timedTransition repairNode3(
    inputs = [nodeOff3],
121 outputs = [node3On],
    delay = 1.0
123 );

125 timedTransition repairNode4(
    inputs = [node4Off],
127 outputs = [node4On],
    delay = 1.0
129 );

131 timedTransition repairVm(
    inputs = [vmOff],
133 outputs = [vmOn],
    inhibitors = [nodeOff],
135 delay = 1.0
    );
137
    timedTransition repairVm2(
139 inputs = [vm2Off],
    outputs = [vm2On],
141 inhibitors = [node2Off],
    delay = 1.0
143 );

145 timedTransition repairVm3(
    inputs = [vm3Off],
147 outputs = [vm3On],
    inhibitors = [nodeOff3],
149 delay = 1.0
    );
151
    timedTransition repairVm4(
153 inputs = [Vm4Off],
    outputs = [vm4On],
155 inhibitors = [node4Off],
    delay = 1.0
157 );

159 metric Availability = stationaryAnalysis( expression = "P{((#frontOn=1)AND(((#nodeOn
        =1)AND(#vmOn>0))OR((#node2On=1)AND(#vm2On>0))OR(#node3On=1)AND(#vm3On>0))OR(#
        node4On=1)AND(#vm4On>0))}" );
```

```
    }
161
    main {
163 Availability = solve( Model,Availability );
    println(Availability);
```