Pós-Graduação em Ciência da Computação

"Análise de Disponibilidade e Consumo Energético em Ambientes de *Mobile Cloud Computing*"

Por

Danilo Mendonça Oliveira

Dissertação de Mestrado

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

RECIFE, Fevereiro/2014

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

Danilo Mendonça Oliveira

"Análise de Disponibilidade e Consumo Energético em Ambientes de *Mobile Cloud Computing*"

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

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

RECIFE, Fevereiro/2014

Dedico esta dissertação ao Rei dos Reis, Jesus Cristo.
Maranata!

Agradecimentos

Agradeço em primeiro lugar ao Criador, pelo dom da vida e por ter me capacitado e provido os recursos necessários para que eu completasse essa etapa da minha vida.

Agradeço ao meu orientador Paulo Maciel que me proporcionou esta oportunidade de crescer em minha carreira acadêmica. Dedico outro agradecimento ao meu "orientador emérito" Ricardo Salgueiro que me ajudou a ingressar no mestrado acadêmico da UFPE e me apresentou ao meu orientador atual.

Agradeço aos meus pais, os quais foram os melhores pais que eu conheci nessa vida, sem sombra de dúvidas. Sempre fizeram o melhor que podiam para dar suporte à minha educação, mesmo nas situações mais adversas.

Agradeço profundamente à minha irmã Dani e meu cunhado Victor por me acolher tão bem em sua casa, me oferecendo abrigo e todo tipo de ajuda possível.

Aos meus amigos, primos, tios e tias e meus queridos avós, que estão em Sergipe, os quais eu sinto uma saudade imensa.

Aos meus novos amigos de Recife, do Centro de Informática da UFPE e da Igreja Presbiteriana da Encruzilhada.

If the automobile had followed the same development cycle as the computer, a Rolls-Royce would today cost $100, get a million miles per gallon, and explode once a year, killing everyone inside.

—ROBERT X. CRINGELY

Resumo

Mobile cloud computing (MCC) é um novo paradigma computacional que tem como objetivo melhorar a capacidade de dispositivos móveis através do provisionamento de recursos virtualizados de uma infraestrutura de nuvem. Enquanto que a MCC melhora consideravelmente as habilidades de tais dispositivos, também impõe a dependência em tempo integral de uma conexão sem fio com a Internet. Além disso, questões como descarga de bateria, falhas no dispositivo móvel, *bugs* de aplicação e interrupções no serviço em nuvem, podem representar obstáculos na disseminação deste paradigma. Sendo um paradigma tão recente, poucos esforços foram feitos no sentido de estudar os impactos destes tipos de falhas sobre atributos de dependabilidade. Desta forma, nosso trabalho tem como proposta oferecer uma metodologia de avaliação de disponibilidade em arquiteturas de *mobile cloud*. A metodologia é dividida em três partes. Primeiro, definimos uma arquitetura base, onde não há mecanismos de redundância e tolerância à falhas. Avaliamos a disponibilidade desta arquitetura por meio de um modelo hierárquico composto de diagramas de blocos de confiabilidade (RBD) e cadeias de Markov de tempo contínuo (CTMC), e validamos o modelo através de um *testbed* de injeção de falhas e por simulação. Na segunda parte, apresentamos três adaptações do cenário base que tem como objetivo o aumento da disponibilidade estacionária. Estes três cenários são avaliados em termos de disponibilidade e *downtime anual*, através de extensões do modelo previamente validado. Por último, selecionamos uma das arquiteturas e realizamos uma investigação mais detalhada dos efeitos do uso de interfaces *wireless* sobre o consumo energético do dispositivo e seu impacto na disponibilidade, utilizando modelos em redes de Petri estocásticas. Nossos resultados mostram a efetividade da arquitetura *cloudlet* na melhoria da disponibilidade do sistema em comparação ao cenário base. Também concluímos que em aplicações móveis que se conectam à nuvem através de múltiplas interfaces de rede (3G e WiFi), melhorias na estabilidade do sinal WiFi promovem um aumento de disponibilidade significativo, além de aumentar o tempo de autonomia da bateria do dispositivo.

Palavras-chave: Computação em Nuvem, Computação Móvel, Modelos Analíticos, Disponibilidade, Avaliação de Consumo Energético

Abstract

Mobile cloud computing (MCC) is a novel computational paradigm that aims at improving the computing power of mobile devices through the provisioning of virtualized resources from a cloud infrastructure. While MCC substantially expands the abilities of such gadgets, it also enforces a full-time dependency on wireless Internet connection. Furthermore, issues such as battery charge depletion, mobile device faults, application bugs, and outages in the cloud service may represent obstacles in diffusion of MCC paradigm. For being a such recent paradigm, few efforts were conduced to evaluate the impact of those types of faults in dependability attributes. In this way, our work has as purpose to offer a methodology for availability evaluation of mobile cloud architectures. This methodology is divided in three parts. First, we defined a basic architecture, where there are no redundancy and fault tolerance mechanisms. This architecture was evaluated by a hierarchical model composed by realibility diagram models (RBD) and continuous time Markov chains (CTMC), and validated by a fault injection testbed, and by simulation. In the second part, we presented three adaptations of the baseline scenario that has as objective the improvement of steady-state availability. Those three scenarios were evaluated by extensions of the previously validated model in terms of availability and annual downtime. Lastly, we selected one of those architectures and conduced a more detailed investigation of the effects of wireless connections on the energy consumption and its impact on availability, using stochastic Petri net models. Our results showed the effectiveness of cloudlet architecture on the availability improvement, when compared to the baseline scenario. We also concluded that on mobile applications that connect to an cloud through multiple network interfaces (3G and WiFi), improvements on the WiFi signal stability promote a significant increase on availability, and additionally increase the battery lifetime of the device.

Keywords: Cloud Computing, Mobile Computing, Analytical Modeling, Availability, Energy Consumption Evaluation

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

ADB *Android Debug Bridge*

AP *Access Point*

API *Application Programming Interface*

BYOD *Bring Your Own Device*

CC Controlador de *Cluster* - *Cluster Controller*

CLC Controlador da Nuvem - *Cloud Controller*

CTMC *Continuous-time Markov chains*

DaaS *Data as a Service*

FSC *File Based Storage Controller*

GSPN *Stochastic Petri Nets*

IaaS *Infrastructure as a Service*

IM *Infrastructure Manager*

KVM *Kernel-based Virtual Machine*

MBaaS *Mobile Backend as a Service*

MCC *Mobile Cloud Computing*

MTTF *Mean Time to Failure*

MTTR *Mean Time to Repair*

NAS *Network-Attached Storage*

NC Controlador do Nó - *Node Controller*

PaaS *Platform as a Service*

PN *Petri Nets*

RBD *Reliability Block Diagram*

S3 *Simple Storage Service*

SaaS *Software as a service*

SC Controlador de Armazenamento - *Storage Controller*

SLA *Service Level Agreement*

SM *Storage Manager*

VM *Virtual Machine*

WAN *Wide Area Network*

WLAN *Wireless Local Area Network*

1

Introdução

O número de usuários de "dispositivos inteligentes" (*tablets*, *smartphones*) vem crescendo a cada ano. Está previsto que em 2014, o número de usuários de smartphones alcance um total de 1,75 bilhões (Boyle 2014). O crescimento deste segmento, aliado à evolução das tecnologias de telecomunicação sem fio, resultou na migração do acesso à *Web* de *desktops* tradicionais para dispositivos móveis. É estimado que em 2014, o número de acessos à *Web* através de dispositivos móveis irá ultrapassar o número de acessos via *desktops* (Meeker 2012). Ao usar *smartphones* e *tablets* como principais dispositivos de acesso à *Web*, os usuários irão exigir o mesmo nível de complexidade das aplicações *desktop* em aplicações móveis (Dinh et al. 2011). Entretanto, tais dispositivos sofrem de escassez de recursos como poder de processamento, memória, armazenamento secundário, largura de banda e provisionamento energético por meio de bateria, o que pode inviabilizar o desenvolvimento de aplicações móveis mais sofisticadas (Qi & Gani 2012).

Em paralelo ao crescimento da computação móvel, a computação nas nuvens vem ganhando destaque e popularidade nos últimos anos. A computação em nuvens é capaz de prover recursos computacionais como servidores, redes, armazenamento, aplicações e serviços, de forma flexível e ágil e com menos esforço de gerenciamento ou interação com o provedor de serviços (Mell & Grance 2011). Uma característica chave da computação nas nuvens é a abundância de recursos computacionais providos para o usuário sob demanda, através do uso de virtualização.

Nesse contexto, surge o paradigma da *Mobile Cloud Computing* (MCC) (Kumar & Lu 2010), que tem como objetivo prover recursos computacionais da nuvem para estender a capacidade de dispositivos móveis. Uma das principais técnicas da *mobile cloud computing* é o *offloading* de aplicações. Esta técnica consiste no particionamento de uma tarefa em componentes que podem executar localmente no dispositivo móvel, ou em recursos virtualizados na nuvem (Kumar & Lu 2010). A ideia é que segmentos

da aplicação que exigem processamento intenso sejam movidos para nuvem, enquanto que tarefas leves sejam executadas no próprio dispositivo. Com o *offloading*, podemos ter aplicações mais inteligentes nas pontas dos dedos do usuário e, adicionalmente, economizar energia da bateria.

Se espera que o mercado da *mobile cloud computing* gere 45 bilhões de dólares em receitas até 2016 (Koopman 2012). Considerando este nível de impacto financeiro, será essencial prover serviços que possam ser justificadamente confiados, isto é, serviços **dependáveis**[1]. A dependabilidade de um sistema está relacionada a uma série de requisitos não funcionais que ele deverá garantir para obter satisfação e fidelidade dos usuários. Dependabilidade é um conceito que engloba vários atributos, tais como: confiabilidade, disponibilidade, segurança, confidencialidade, integridade e manutenibilidade (Avizienis et al. 2001). Dentre os atributos de dependabilidade, a **disponibilidade** é o que se relaciona diretamente com a proporção de tempo que o sistema é encontrado operacional. Alcançar alta disponibilidade é importante para evitar perda de lucros e outras consequências desastrosas da indisponibilidade de um serviço.

Atingir níveis satisfatórios de dependabilidade poderá se tornar um desafio em ambientes de *mobile cloud*, uma vez que clientes móveis operam em dispositivos alimentados por bateria e conectados via redes *wireless*. Descarga da bateria e bloqueio do sinal *wireless* são os principais problemas que poderão afetar a disponibilidade de uma aplicação que executa em um dispositivo móvel. Além disso, há um efeito combinado entre esses dois fatores, uma vez que o uso de uma interface de rede *wireless* provoca um aumento no consumo energético da aplicação. Quando consideramos um cenário com múltiplas redes, geralmente a que possui o melhor desempenho, também é a que poupa mais energia da bateria (Balasubramanian et al. 2009, Ra et al. 2010).

Este trabalho propõe uma metodologia para a avaliação de disponibilidade e consumo energético de ambientes de *mobile cloud*. A primeira etapa da metodologia consiste na avaliação de disponibilidade de uma arquitetura básica de *mobile cloud* através de modelos RBD (*Realibility Block Diagram*) (Rausand & Høyland 2003) e CTMC (*Continuous Time Markov Chain*) (Kleinrock 1975). Este cenário é também avaliado por experimento de injeção de falhas e por simulação, a fim de validar o modelo analítico. Para realizar o experimento, construímos um *testbed*[2] composto por: 1) um protótipo de um sistema de móvel em nuvem; 2) um injetor de falha capaz de atuar nos componentes da nuvem, no cliente móvel e na conexão *wireless*; 3) um monitor de disponibilidade; 4) um gerador de

[1]O adjetivo "dependável" não existe na língua portuguesa, embora exista o termo "dependable" em inglês. Este termo caracteriza um sistema que satisfaz as exigências sobre os atributos de dependabilidade.

[2]Plataforma ou conjunto de ferramentas para testes e experimentação sobre um sistema

workload. Uma vez que temos o modelo validado para a arquitetura básica, nós propomos extensões desta arquitetura com o objetivo de investigar melhorias que promovam um aumento da disponibilidade do sistema. As extensões para o modelo básico são: 1) arquitetura *store and forward*; 2) múltiplas interfaces de rede; 3) arquitetura *cloudlet* (Satyanarayanan et al. 2009). Também investigamos o impacto de mecanismos de redundância na infraestrutura de nuvem sobre a disponibilidade e o *downtime* anual. Os cenários considerados anteriormente foram novamente avaliados com a adição deste mecanismo, a fim de avaliar se há uma melhoria significativa nestes atributos através desta funcionalidade.

Para avaliar o impacto da conectividade sem fio sobre o consumo energético e disponibilidade, e a interdependência entre estes dois aspectos, criamos um modelo de disponibilidade mais elaborado da arquitetura de *cloudlet* em rede de Petri estocástica (Chiola et al. 1993). A partir deste modelo, geramos outro modelo focado em consumo energético, que nos permite avaliar o tempo de vida da bateria, considerando o uso combinado de redes WiFi e 3G.

1.1 Motivação e Justificativa

Escassez de recursos computacionais é um dos principais problemas enfrentados por usuários e desenvolvedores de aplicações móveis (Satyanarayanan et al. 2009). Até mesmo com os avanços no *hardware* dos *smartphones* mais modernos - com processadores de quatro núcleos, dois *gigabytes* de memória principal, 32 *gigabytes* de armazenamento secundário - estas configurações serão sempre inferiores às de *desktops* da mesma geração. Ademais, se projetarmos uma aplicação tão complexa que seja adequada apenas para estes dispositivos mais poderosos, nós restringiremos a aplicação apenas para os donos destes aparelhos mais caros. Usuários de *smartphones* mais simples e de *feature phones* [3] serão incapacitados de acessar esta aplicação. Também devemos levar em consideração que a tecnologia de baterias não evoluíram no mesmo ritmo que outros componentes de *hardware* (Cuervo et al. 2010), portanto, aplicações mais pesadas poderão consumir um percentual significativo da carga da bateria.

Embora uma intenção da MCC seja salvar energia através do *offloading* de aplicação, o consumo de energia permanece ainda como um fator crítico em aplicações móveis. A descarga completa da bateria pode interromper o provisionamento de um serviço em

[3] O termo "*feature phones*" designa telefones celulares mais simples e baratos que *smartphones*, com menos recursos e opções de aplicativos

algum momento crítico para o usuário. Além disso, ao mover dados e processamento para a nuvem, a MCC impõe a exigência de uma conexão de dados sempre ativada. Como conexões sem fio são vulneráveis a muitos fatores externos como: interferência eletromagnética, bloqueio do sinal por objetos físicos, *handoff*[4], etc., isto implica em um ponto fraco na disponibilidade do sistema (Chen et al. 2003).

Um sistema em *mobile cloud computing* que falhe em oferecer níveis satisfatórios de disponibilidade, segurança e integridade de dados, poderá oferecer uma péssima experiência para o usuário e causar prejuízos para o provedor da aplicação. Uma aplicação móvel disponível em uma loja *online* está sujeita a comentários e avaliações dos usuários. Desta forma, um sistema móvel mal projetado poderá ser afetado por críticas severas, levando a danos à reputação do produto e de seus criadores.

A *mobile cloud computing* também vem sendo empregada por áreas de aplicações críticas como a telemedicina (Doukas et al. 2010, Hoang & Chen 2010, Varshney 2007, Tang et al. 2010). Neste caso, a disponibilidade e confiabilidade do serviço serão atributos essenciais. Por exemplo, considere uma aplicação de telemedicina que monitora sinais vitais de um paciente em casa e envia-os a um médico através da Internet. Uma falha na provisão do serviço poderá significar que o paciente não será monitorado por algum período de tempo. Com isso, o médico poderá não ser notificado de alguma condição crítica do paciente, implicando em riscos à saúde deste.

1.2 Trabalhos Relacionados

Até o momento, não encontramos na literatura um trabalho que faça um uso combinado de modelos analíticos e injeção de falhas para avaliar a disponibilidade de ambientes em *mobile cloud computing*. Desta forma, os trabalhos relacionados podem ser classificados em duas categorias: avaliação de disponibilidade de *mobile cloud computing* e mecanismos de injeção de falhas em dispositivos móveis.

Courtes et al. (2007) apresentam um estudo de características de dependabilidade em um serviço de *backup* colaborativo *ad hoc* para dispositivos móveis, baseado na plataforma MoSAIC (Killijian et al. 2004). Esta plataforma utiliza a técnica de *Erasure Codes* (Lin et al. 2004) para aumentar a confiabilidade e disponibilidade dos dados. Esta técnica consiste em fragmentar uma entrada de dados formada por k símbolos em n fragmentos que são espalhados pelos dispositivos que colaboram no serviço através de

[4]O termo refere-se ao processo de migração da área de cobertura de um ponto de acesso/estação base *wireless* para outra área de cobertura de outro ponto de acesso/estação base

conexões *ad hoc*. Quando algum dispositivo faz uma conexão com a internet, ele envia os fragmentos para o serviço de armazenamento em nuvem. A arquitetura é avaliada através de redes de Petri estocásticas generalizadas (GSPN - Generalized Stochastic Petri Nets) (Chiola et al. 1993), fazendo algumas simplificações: i) a nuvem é 100% confiável; ii) quando um dispositivo faz conexão com a internet, essa conexão tem largura de banda suficiente para enviar todos os fragmentos daquele dispositivo de uma só vez. Os resultados mostram a efetividade da arquitetura MoSAIC em aumentar a disponibilidade, contudo, a técnica de *Erasure Codes* tem vantagens sobre a replicação simples (sem fragmentação dos dados) apenas em um número limitado de cenários.

O problema da exigência de uma conexão de dados sempre ativada é reconhecido por Klein et al. (2010). Esse trabalho propõe uma arquitetura de acesso inteligente baseada em informações de contexto gerenciadas pela *mobile cloud computing* como forma de otimizar o acesso móvel em um ambiente altamente heterogêneo, levando em conta eficiência energética e custo. O artigo apresenta a proposta da arquitetura, e um simulador de rede para validar o trabalho. Entretanto, não é mostrado nenhum tipo de resultado experimental, sendo deixado como trabalho futuro. Além disso, não encontramos na literatura a continuação deste trabalho.

Os efeitos do uso combinado das interfaces *wireless* WiFi e 3G são investigados em Lee et al. (2010). Este trabalho considera duas abordagens para a sincronização de dados com a nuvem: *On-the-spot*, na qual o usuário inicia a transmissão dos dados via WiFi, e usa uma conexão 3G para transmitir o resto dos dados quando o usuário sai fora da zona de cobertura; *Delayed offloading*, na qual o usuário interrompe a transmissão dos dados quando sai fora da zona de cobertura WiFi, e transmite o restante apenas quando um certo *deadline* é atingido. Os autores concluíram que a estratégia de *offloading* "*On-the-spot*", que é o padrão adotado por muitos usuários, consegue uma redução no tráfego de dados 3G de até 65%, e uma redução de energia de 55%. A técnica de "*Delayed offloading*" só consegue atingir bons resultados sobre a estratégia anterior, quando o intervalo de *delay* é de uma hora ou mais.

Pandey & Nepal (2012) proporam uma métrica de disponibilidade definida em função de um perfil de disponibilidade, modelo de custo, e utilização de serviço. Este trabalho faz uma análise da disponibilidade do serviço em nuvem e sua relação com o custo, levando em consideração a utilização do serviço por um grande número de usuários. Comparado com o nosso trabalho, este trabalho possui uma visão mais alto nível da disponibilidade do serviço em nuvem, uma vez que abstrai os componentes do lado da nuvem. Em contrapartida, o nosso trabalho analisa a disponibilidade do ponto de vista do

cliente móvel, levando em consideração falhas no dispositivo móvel, na conexão sem fio, além de modelar componentes de redundância do lado da nuvem.

Em Acker et al. (2010) é proposto o porte de um injetor de falhas existente, o FIRMAMENT (Drebes 2005), para a plataforma *Android* (Android 2013). O FIRMAMENT é um injetor de falhas que funciona a nível de *kernel*, capaz de injetar falhas de comunicação em fluxos de comunicação TCP e UDP. Este injetor de falha é capaz de introduzir os seguintes erros em mensagens: atraso, duplicação , corrupção do conteúdo e descarte. Apesar de o sistema *Android* ser uma versão modificada do Linux 2.6, foi possível o porte da ferramenta FIRMAMENT para o sistema *Android*. Entretanto, foi necessário compilar o kernel do *Android* incluindo o módulo *Netfilter*, que permite a inserção de ganchos de sistema operacional nas mensagens recebidas.

O trabalho proposto por Cinque et al. (2012) consiste em uma plataforma de *logging* de eventos de falha em smartphones *Android*. Este esta plataforma é uma extensão de uma plataforma anterior construída pelos autores para a plataforma *Symbian* (Cinque et al. 2007). Em contraste com a natureza de código fechado do *Symbiam*, o sistema *Android* permite uma maior introspecção dos eventos a nível de sistema operacional, permitindo que um maior conjunto de falhas seja observado. A plataforma de log é capaz de registrar 4 tipos diferentes de falhas: bugs de aplicação, congelamento do celular, *auto-reboot* e travamento da aplicação. Os autores também construíram uma plataforma de injeção de falhas para validar a plataforma de *logging*. Infelizmente, ainda não foram publicados testes com a plataforma desenvolvida. Estes testes poderiam prover informações úteis para o trabalho atual, uma vez que não conseguimos encontrar na literatura tempos de falha para aplicativos, SO móvel e dispositivos.

1.3 Objetivos

O objetivo principal deste trabalho é propor uma metodologia de avaliação de disponibilidade para ambientes de *mobile cloud computing*, combinando o uso de modelos analíticos, modelos de simulação e experimentos de injeção de falhas. A metodologia é aplicada em uma série de cenários de *mobile cloud*, com o objetivo de avaliar:

- O impacto de fatores como descarga da bateria, conectividade *wireless*, falhas no sistema móvel e falhas no serviço em nuvem sobre a disponibilidade;
- O impacto de mecanismos de redundância na arquitetura de uma *mobile cloud* no aumento da disponibilidade do sistema.

A seguir, apresentamos a lista de objetivos secundários deste trabalho:

- A construção de um *testbed* de injeção de falhas para ambientes de *mobile cloud*;
- A criação de uma API de simulação para avaliação de dependabilidade, que pode ser utilizada para a validação de modelos analíticos e experimentos, e também para a modelagem de cenários cujos tempos entre falhas e reparos não seja exponencialmente distribuídos;
- A elaboração de modelos para a avaliação da disponibilidade de arquiteturas de *mobile cloud*;
- A elaboração de modelos para avaliação do consumo energético em dispositivos móveis, levando em consideração o uso de interfaces *wireless*;
- Discutir os *tradeoffs* envolvidos entre diferentes alternativas de arquiteturas para *mobile cloud*, e efetividade destas no aumento da disponibilidade.

1.4 Estrutura da Dissertação

O Capítulo 2, apresenta a fundamentação teórica do trabalho: *mobile cloud computing*, avaliação de dependabilidade e injeção de falhas.Antes de introduzir os conceitos sobre *mobile cloud*, apresentamos os dois paradigmas que formam a MCC: a computação em nuvens e a computação móvel. Em seguida, fazemos uma discussão sobre dependabilidade, em particular, sobre o conceito de disponibilidade que é crucial para este trabalho. Após essa discussão, explanamos sobre as técnicas de modelagem que nos permite realizar a avaliação de atributos de dependabilidade. Os formalismos empregados foram: 1) diagramas de blocos de confiabilidade (RBD), 2) cadeias de Markov de tempo contínuo (CTMC) e redes de Petri estocásticas generalizadas (GSPN). Por último, apresentamos a técnica de injeção de falhas para avaliação de disponibilidade.

O Capítulo 3 introduz a metodologia proposta neste trabalho. Inicialmente ele descreve a metodologia em linhas gerais e, em seguida, detalha as etapas preliminares. O cenário básico de uma arquitetura de *mobile cloud* é apresentado neste capítulo, seguido do *testbed* de injeção de falhas e a API de simulação que foram desenvolvidos.

O Capítulo 4 apresenta algumas alternativas arquiteturais de *mobile cloud* e seus respectivos modelos de disponibilidade. Além disso, o capítulo propõe modelos em redes de Petri para a análise de disponibilidade e consumo energético. O Capítulo 5 mostra os estudos de caso desta dissertação, que são baseados nos modelos propostos. Inicialmente,

apresentamos a validação do modelo analítico através do experimento conduzido com o *testbed* de injeção de falhas e pela execução do modelo de simulação. Nos estudos de casos descritos posteriormente, investigamos a melhoria da disponibilidade da arquitetura base de *mobile cloud* causada pelas alternativas arquiteturais consideradas e por um mecanismo de redundância no serviço em nuvem. Apresentamos também um estudo de caso para investigar o efeito do uso de conexões *wireless* sobre a disponibilidade e consumo energético.

Finalmente, o Capítulo 6 conclui este trabalho, apresentando as principais contribuições e sugerindo possíveis trabalhos futuros.

2

Fundamentação Teórica

Este capítulo apresenta os conceitos básicos sobre os temas abordados nesta dissertação e está dividido em duas partes. Na primeira parte, expomos os fundamentos em *Mobile Cloud Computing*, acompanhado de um breve resumo dos principais trabalhos encontrados na literatura. Na segunda parte, explicamos os conceitos básicos sobre dependabilidade e as técnicas de avaliação de dependabilidade utilizadas neste trabalho. Em seguida, apresentamos os formalismos de diagramas de blocos de confiabilidade, cadeias de Markov de tempo contínuo e redes de Petri estocásticas. Após descrever os formalismos, apresentamos os *softwares* utilizados para a concepção e avaliação dos modelos utilizados neste trabalho. Por último, explanamos sobre técnicas de injeção de falhas para avaliação de dependabilidade.

2.1 Fundamentos em *Mobile Cloud Computing*

Nesta seção, iniciamos com um resumo sobre os dois pilares da *Mobile Cloud Computing*: computação em nuvens e a computação móvel e, em seguida, falamos do tema principal.

2.1.1 *Cloud Computing*

Computação em nuvens é um modelo computacional no qual recursos como poder de processamento, rede, armazenamento e softwares são oferecidos através da Internet e podem ser acessados remotamente (Armbrust et al. 2010). Este modelo permite que usuários obtenham os recursos de forma elástica, sob demanda e a um baixo custo, entregues de maneira semelhante a serviços tradicionais como água, gás, eletricidade e telefonia (Dinh et al. 2011). Dentre as inúmeras definições de *cloud computing* existentes na literatura, apresentamos a definição proposta por Vaquero et al. (2008):

> *Clouds são um grande conjunto de recursos virtualizados facilmente usáveis e acessíveis (tais como hardware, plataforma de implantação e/ou serviços). Estes serviços podem ser dinamicamente reconfigurados para uma carga variável, permitindo uma utilização ótima dos recursos. Este conjunto de recursos é explorado tipicamente por um modelo pague-por-uso no qual garantias são oferecidas pelo provedor da infraestrutura por meio de SLAs (Acordo de Nível de Serviço) customizados.*

A computação em nuvem é um paradigma computacional recente, mas que foi construído com base em tecnologias existentes como virtualização, *grid computing* e *utility computing*. Os recursos computacionais de uma *grid* são provisionados para os clientes como VMs através da virtualização de uma infraestrutura de *datacenter*, de forma que cada cliente só paga pela quantidade de recursos que consumir, em vez de pagar uma taxa fixa (*utility computing*).

Este modelo computacional tem várias características que o torna atrativo para um grande número de corporações. Se uma empresa deseja lançar um serviço na Internet, ela pode simplesmente alugar um conjunto de VMs em um serviço na nuvem e hospedar seu serviço nela, sem a necessidade de manter sua própria infraestrutura de servidores (Zhang, Cheng & Boutaba 2010). Isso ajuda a reduzir o "*time to market*" de seus produtos e, consequentemente, tornando-os mais competitivos. Além disso, a propriedade de provisionamento de recursos de forma elástica de uma *cloud* permite que o sistema se adapte a mudanças abruptas no volume da carga de trabalho, sem a necessidade de um superdimensionamento da infraestrutura usada. Como exemplo, podemos considerar um portal de serviços acadêmicos (igual ao Sig@ da UFPE [1]). Em épocas de matrícula e vestibular, o volume de acessos aumenta drasticamente em comparação ao resto do ano. Um administrador de tal sistema poderia alocar um maior número de VMs mais poderosas durante esta época do ano, de forma a atender a esta grande demanda, e utilizar VMs mais modestas no resto do tempo, quando a quantidade de acessos fosse menor. A grande vantagem é que provedor de serviço não precisa superdimensionar uma infraestrutura de servidores para suportar os picos na carga de trabalho, que como consequência vai ficar subutilizada durante a maior parte do tempo. Para exemplificar este fato, a *Amazon* - que é uma das grandes corporações do segmento da *cloud computing* nos dias de hoje - usava em torno de 10% da capacidade do seu *datacenter*, que foi dimensionado para suportar momentos de pico não muito frequentes (Hof 2006).

[1] https://www.siga.ufpe.br/ufpe/index.jsp

Podemos classificar as nuvens computacionais sobre dois aspectos. O primeiro aspecto é relativo ao ponto de vista do usuário da nuvem e quem mantém a infraestrutura. Sobre esse aspecto, podemos classificar uma nuvem como (Zhang, Cheng & Boutaba 2010):

- **Privada**: São infraestruturas de nuvem projetadas para serem usadas por uma única organização. Uma infraestrutura de nuvem privada pode ser provida por um agente externo, ou ser gerenciada pela própria empresa. No último caso, essa opção é optada por organizações que possuem dados muito críticos (bancos, órgãos do governo, agências militares, etc.), não permitindo que estes dados sejam confiados a terceiros (no caso, provedores de nuvens públicas). A desvantagem dessa abordagem em relação a optar por nuvens públicas é a exigência do investimento inicial em uma *server farm* proprietária (Zhang, Cheng & Boutaba 2010), igual ao modelo tradicional sem uso de *cloud*. Entretanto, ainda contamos com as vantagens da consolidação e gerenciamento dos recursos de uma organização, permitindo um uso mais eficiente destes (Zhao & Figueiredo 2007, Kim et al. 2009);

- **Pública**: São nuvens mantidas por provedoras de serviço, que disponibilizam seus recursos computacionais para outras organizações. Este tipo de nuvem reflete mais os benefícios originais propostos pela computação em nuvens do que as nuvens privadas. Diferente de uma nuvem privada, o investimento inicial de infraestrutura é zero, pois não é necessário adquirir servidores e outros equipamentos próprios. Nuvens privadas geralmente são oferecidas por grandes *datacenters* que possuem uma capacidade computacional bastante superior ao de uma nuvem privada. Assim, se o volume de acessos a um serviço aumentar drasticamente, é possível alocar mais recursos virtualizados para se adaptar à nova carga de trabalho. Outra vantagem adicional é delegar para o provedor da nuvem a responsabilidade pelo gerenciamento da infraestrutura e por manter o SLA combinado (Carolan et al. 2009). A desvantagem deste tipo de nuvem é a falta de controle sobre dados, rede e configurações de segurança, o que impede algumas corporações de utilizar este tipo de nuvem, sobretudo as que lidam com dados extremamente confidenciais, citadas anteriormente;

- **Híbrida**: É uma combinação dos modelos de nuvem pública e privada, a fim de superar as limitações de cada abordagem. A vantagem deste modelo é proporcionar o melhor dos dois mundos. Dados críticos de uma organização podem ser mantidos na parte privada da nuvem, ao mesmo tempo em que é possível escalar o serviço

através da imensa capacidade de uma nuvem privada. Os benefícios vem acompanhados de uma maior complexidade no gerenciamento da nuvem, e determinar o particionamento ótimo entre os componentes públicos e privados pode ser um desafio (Zhang, Cheng & Boutaba 2010).

O outro aspecto sobre o qual as nuvens computacionais podem ser classificadas é quanto seu modelo de negócio:

- **Infraestrutura como Serviço (IaaS)** - Através do uso da virtualização, este modelo de negócio particiona os recursos de um datacenter entre os usuários sobre a forma de recursos virtualizados. Neste modelo de negócio, geralmente o usuário paga pelo serviço de acordo com a capacidade da VM alocada e pela quantidade de tempo que a VM permanece em execução. O usuário é responsável por manter sua própria pilha de software na qual os seus serviços irão executar;
- **Plataforma como Serviço (PaaS)** - Neste modelo de negócio, o provedor de serviço abstrai a VM e o sistema operacional utilizado, e oferece ao usuário uma plataforma de *software* de alto nível. Este modelo tem como vantagem o dimensionamento transparente dos recursos de *hardware* durante a execução dos serviços (Vaquero et al. 2008);
- **Software como Serviço (SaaS)** - Este modelo de negócio é voltado para usuários comuns de internet, em vez de desenvolvedores e administradores de sistema. Neste modelo, softwares e dados gerados por eles ficam armazenados na nuvem e podem ser acessados a partir de qualquer computador conectado à internet através de uma interface *Web*.

Além destes principais modelos descritos acima, também existe o modelo de *Data as a Service* (DaaS) (Truong & Dustdar 2009), no qual a nuvem oferece ao seus clientes um serviço de armazenamento secundário, garantindo alta disponibilidade e integridade dos dados. Um serviço de DaaS pode ser visto como IaaS ou como SaaS, dependendo do contexto sob o qual é utilizado. Considere como exemplo o serviço de armazenamento e backup em nuvem do *Dropbox*. O serviço que o *Dropbox* (Dropbox 2013) oferece para seus usuários pode ser enxergado como SaaS ou DaaS. Contudo, o próprio *Dropbox* utiliza o serviço de armazenamento da *Amazon* - o S3 (*Simple Storage*) (AmazonS3 2013) - para armazenar os dados dos seus clientes. Do ponto de vista do *Dropbox*, o serviço de armazenamento da *Amazon* pode ser considerado como IaaS.

2.1.2 Computação Móvel

O termo "computação móvel" é bastante abrangente, e engloba diversos cenários que fogem do âmbito desta dissertação. Em vez de fazer uma discussão extensa e generalista sobre a computação móvel, iremos delimitar o escopo da computação móvel para o considerado pela *mobile cloud computing*. Por exemplo, um cenário onde usuários de *notebooks* acessam serviços em nuvem de diferentes locais, através *hotspots* WiFi e modem 3G, é de fato um cenário de computação móvel. Contudo, a *mobile cloud computing* não tem como objetivo atender este tipo de usuário móvel, mas sim usuários de dispositivos menores como celulares, *smartphones* e *palms*, que possuem maiores limitações de capacidade computacional do que notebooks.

Definimos a computação móvel a partir de duas propriedades principais. A primeira é que os dispositivos devem ser portáteis, de forma que suas dimensões reduzidas permitam seus usuários permanecerem com tais dispositivos a maior parte do tempo. Esta propriedade é abreviada pela sigla BYOD *(Bring Your Own Device)* (Archer et al. 2012). A segunda propriedade crucial da computação móvel é a **conectividade**, isto é, a capacidade de receber e enviar dados para a Internet através de redes sem fio.

Os requisitos que existem sobre os dispositivos móveis em relação a tamanho, peso, ergonomia, etc. implicam em restrições nas capacidades de tais dispositivos. Usuários esperam aplicações móveis mais sofisticadas, que necessitam de dispositivos com processadores mais rápidos, além de telas maiores e com maior resolução. Infelizmente isto implica em um aumento significativo no consumo energético. A seguir, listamos as principais limitações da computação móvel (Satyanarayanan 1996) :

- **Aparelhos móveis serão sempre mais restritos em recursos do que aparelhos estáticos** – É notável que a capacidade dos aparelhos de hoje evoluiu significativamente ao longo dos anos, superando computadores *desktops* de alguns anos atrás. Contudo, as restrições inerentes à mobilidade e portabilidade impedem que seu poder seja comparável aos *desktops* da mesma época. Mesmo os aparelhos mais modernos de hoje possuem uma capacidade de processamento três vezes menor, memória principal oito vezes menor, armazenamento secundário de cinco a dez vezes menor, e largura de banda dez vezes inferior, aos *desktops* atuais (Qi & Gani 2012).

- **Mobilidade é inerentemente perigosa** – A natureza portátil de um *smartphone* também implica em certos riscos relacionados à segurança. Um *smartphone* pode ser roubado, perdido ou esquecido em algum lugar. Além do prejuízo financeiro,

podemos ter nossa privacidade invadida, uma vez que deixamos gravados em nossos aparelhos: mensagens particulares, *emails*, lista de contatos, senhas de redes sociais, fotos pessoais, etc. Sistemas operacionais, aplicações e redes móveis também podem ter vulnerabilidades que são exploradas por *hackers*, que tem como objetivo obter acesso a essas informações confidenciais. Nos últimos anos, diversas celebridades foram expostas na mídia porque tiveram fotos íntimas roubadas de seus aparelhos.

- **Conectividade móvel é altamente variável em desempenho e confiabilidade** – A mobilidade do usuário implica que as condições da conectividade *wireless* serão altamente variáveis, uma vez que ele pode sair de uma rede WLAN de alta velocidade e ir para uma rede WAN de baixa velocidade, ou até mesmo permanecer desconectado (Satyanarayanan 1996). O usuário pode desligar a interface de rede voluntariamente, caso o nível de bateria esteja crítico. Além disso, sinais sem fio são mais vulneráveis à interferência e *spoofing*, o que representa uma desvantagem em relação à computação fixa (Qi & Gani 2012).

- **Aparelhos móveis confiam em uma fonte finita de energia** – Este é um dos maiores problemas enfrentados pela computação móvel em geral, sejam *smartphones*, *tablets*, *notebooks*, sensores, etc. A evolução da capacidade dos dispositivos móveis nesta última década veio acompanhada de uma maior demanda de energia. Contudo, a tecnologia das baterias não conseguiu acompanhar esta evolução. Essa é a maior reclamação de pessoas que migram de *feature phones* para *smartphones*. Enquanto a bateria dos *feature phones* duram vários dias sem precisar recarregar (até 5 dias, dependendo do uso), *smartphones* precisam ser recarregados todo dia.

2.1.3 *Mobile Cloud Computing*

O paradigma da *Mobile Cloud Computing* (Kumar & Lu 2010) apareceu nos últimos anos como a intersecção das duas subáreas da TI discutidas anteriormente: computação móvel e a computação nas nuvens. À medida que o mercado da computação móvel cresce, usuários exigem aplicações móveis mais avançadas com o mesmo nível de complexidade oferecida por aplicações *desktop*. Entretanto, até mesmo *smartphones* mais modernos sofrem de limitação de recursos, principalmente CPU, memória, armazenamento, conexão sem fio e bateria. Ao contrário dos dispositivos móveis, uma nuvem possui um grande poder computacional e pode provê-lo sob demanda aos seus usuários.

O objetivo principal da MCC é usar a capacidade de processamento e armazenamento de uma nuvem computacional para estender as capacidades de dispositivos móveis mais limitados, de forma a melhorar o desempenho e aliviar o consumo energético de aplicações móveis mais pesadas. Este novo paradigma poderá permitir o provisionamento de aplicações móveis mais inteligentes, que estendam as capacidades cognitivas de usuários, tais como: reconhecimento de voz, processamento de linguagem natural, visão computacional, aprendizado de máquina, realidade aumentada, planejamento e tomada de decisão (Satyanarayanan et al. 2009).

O encontro entre a computação móvel e a computação em nuvens foi possível graças ao número crescente de usuários com acesso à Internet móvel. De acordo com Meeker (2012), estima-se que em 2014 o número de acessos à Web através de dispositivos móveis irá ultrapassar os acessos feitos por *desktops*. Como consequência, dispositivos móveis poderão ser vistos como principais pontos de entradas e interfaces para *softwares* e serviços na nuvem. Em adição aos modelos de serviço tradicionais da computação em nuvens (Infraestrutura como Serviço, Plataforma como Serviço e *Software* como Serviço), a *mobile cloud computing* trouxe outro modelo de serviço chamado *Mobile Backend as a Service* (MBaaS). Uma plataforma de MBaaS provê funcionalidade básica para aplicações móveis em nuvem, como por exemplo, gerenciamento de usuários, notificações *push* e integração com redes sociais (Sareen 2013).

Na Figura 2.1, descrevemos uma arquitetura típica de uma *mobile cloud*. Podemos visualizar nesta figura que a *mobile cloud computing* é simplesmente a computação em nuvem quando os clientes deixam de ser desktops tradicionais e passam a ser móveis, que se conectam à nuvem via uma infraestrutura de rede sem fio (WLAN, 3G, GPRS, WiMAX, etc). Nesta arquitetura, uma vez que a computação é migrada para a nuvem, os requisitos sobre a capacidade dos dispositivos móveis são diminuídas, e uma gama maior de clientes pode ser atingida, como *smartphones* mais poderosos, *smartphones* mais limitados, *palms* e até mesmo alguns *feature phones* (Qi & Gani 2012). Uma vez que a nuvem passa ser detentora da computação e armazenamento em aplicações móveis, a conexão sem fio passa a ser um elemento chave desta arquitetura. Considerando que redes sem fio são menos estáveis que redes cabeadas usadas por *desktops* tradicionais, isto representa um desafio para a *mobile cloud computing*. Além disso, interfaces *wireless* estão entre os componentes de dispositivos móveis que mais gastam energia. Desta forma, como veremos adiante, a comunicação com a nuvem pode diminuir a economia energética proporcionada pelos mecanismos de *offloading* de uma *mobile cloud*.

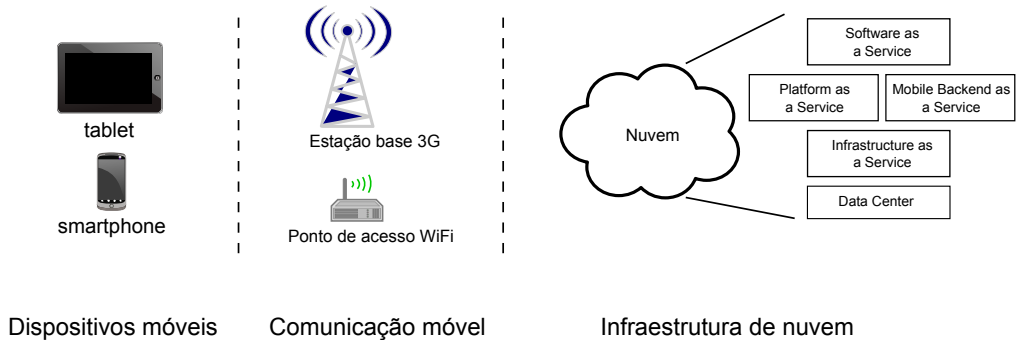


Figura 2.1: Arquitetura de *Mobile Cloud Computing* (adaptada de (Qi & Gani 2012))

2.1.3.1 *Offload* de computação

De acordo com Kumar & Lu (2010), a quantidade de energia economizada por um esquema de *offloading* de aplicação móvel é representada pela Equação 2.1.

$$P_c \times \frac{C}{M} - P_i \times \frac{C}{S} - P_{tr} \times \frac{D}{B} \tag{2.1}$$

, onde:

- C é o número de instruções de uma determinada tarefa computacional;
- S e M são as velocidades, em instruções por segundo, da nuvem e do dispositivo móvel, respectivamente;
- D é o total de dados enviados durante a comunicação com a nuvem;
- B é a largura de banda da conexão entre o cliente móvel e a nuvem;
- P_c é a potência do dispositivo ao realizar computação da tarefa;
- P_i é a potência do dispositivo quando ocioso (quando a computação está sendo realizada na nuvem e o dispositivo aguarda os resultados);
- P_{tr} é a potência do dispositivo ao transmitir dados para a nuvem.

De acordo com a Equação 2.1, a energia salva pelo mecanismo de *offloading* corresponde à diferença entre a energia gasta para computar a tarefa localmente e a energia gasta para delegar a execução da tarefa na nuvem. A energia despendida pela execução local da tarefa é determinada por $P_c \times \frac{C}{M}$. A energia consumida no dispositivo móvel durante a execução remota da tarefa corresponde a soma de: i) a quantidade de energia gasta ao

esperar o término da tarefa na nuvem ($P_i \times \frac{C}{S}$); ii) a energia consumida pela comunicação com a nuvem para a delegação da tarefa e recebimento dos resultados ($P_{tr} \times \frac{D}{B}$).

Podemos perceber que a quantidade de energia que podemos economizar envolve um compromisso entre algumas variáveis. Por exemplo, se uma tarefa tiver um tempo de execução curto, que pode ser consequência do tamanho da tarefa ou de um dispositivo mais poderoso (com processador de quatro núcleos, por exemplo), a quantidade de energia salva pela execução remota ($P_c \times \frac{C}{M}$) pode ser inferior ao *overhead* da comunicação com a nuvem ($P_{tr} \times \frac{D}{B}$). Isto torna o resultado da fórmula negativo, que significa que estamos desperdiçando energia em vez de economizar. Se o *offloading* de uma tarefa causar uma transferência de um conjunto de dados D muito grande, ou se a largura de banda B for muito baixa, esta diferença também poderá ser negativa.

Levando em consideração que as condições da rede *wireless* podem ser variáveis de acordo a mobilidade do usuário e sobrecarga da rede pelo conjunto total de usuários, e que de acordo com uma determinada entrada, uma tarefa pode demorar mais ou menos tempo para executar, concluímos que a decisão de delegar uma tarefa para a nuvem não pode ser feita de maneira estática. Um mecanismo inteligente de *offloading* monitora as condições da rede e do usuário a fim de saber quando é o melhor momento para delegar a execução para a nuvem. Neste contexto, existem diferentes estratégias de *offloading* e arquiteturas de *mobile cloud* que foram propostas nos últimos anos. A seguir, apresentamos as principais arquiteturas encontradas na literatura.

Chun & Maniatis (2009) propõem uma técnica, denominada *Clone Cloud*, baseada na clonagem do sistema operacional móvel como uma máquina virtual na nuvem. O clone terá à disposição mais recursos disponíveis na nuvem, e poderá ser usado para executar tarefas mais intensas. Essa abordagem tem como desvantagem a transferência de todo o estado do sistema móvel através de redes WAN, o que pode resultar em um grande tempo de resposta. Isso significa que o valor de D na fórmula descrita anteriormente pode ser demasiado grande, o que implica em um maior *overhead* e uma diminuição na economia de energia. Uma solução arquitetural para superar este problema é proposta por Satyanarayanan et al. (2009), que consiste em usar uma infraestrutura privada de *cloud* – a *cloudlet* – para oferecer um melhor desempenho e tempo de resposta através da rede local. Nessa abordagem, a principal vantagem é aumentar o peso de B da fórmula acima, uma vez que a vazão de *bits* numa WLAN geralmente é superior a de uma conexão 3G.

A metodologia de *offloading* proposta por Giurgiu et al. (2009) consiste na abstração da aplicação através de um grafo. Sobre este grafo, são executados algoritmos com o objetivo de encontrar o corte maximal, que maximiza ou minimiza uma certa função

objetivo. Esta função objetivo está ligada a alguma meta do usuário, como por exemplo, minimizar a latência ou transferência de dados. Esta abordagem tem como desvantagem o particionamento estático e inflexível da aplicação. Uma abordagem mais flexível que a anterior é o conceito de *weblets* (Zhang, Jeong, Kunjithapatham & Gibbs 2010). *Weblets* são entidades computacionais que podem executar no dispositivo móvel ou em nós da nuvem, de forma transparente. Esta abordagem utiliza um motor de aprendizado para decidir o local ideal para execução de cada *weblet*, considerando fatores como *status* do dispositivo, *status* da nuvem, medidas de desempenho da aplicação e preferências do usuário. Desta forma, um *weblet* pode decidir em tempo de execução o melhor local para executar, de acordo com a situação.

2.2 Dependabilidade

Além de questões de desempenho, existe um conjunto de requisitos não funcionais que um sistema deve alcançar a fim de obter satisfação do usuário. Estes requisitos estão listados abaixo (Avizienis et al. 2001):

- **Confiabilidade** - A probabilidade que o sistema irá prover o serviço continuamente, sem erros, até um certo instante t;
- **Disponibilidade** - A capacidade do sistema estar de prontidão para prover o serviço corretamente;
- **Segurança** - Ausência de consequências catastróficas para o(s) usuário(s) e o ambiente;
- **Confidencialidade** - Ausência de divulgação desautorizada de informação;
- **Integridade** - Ausência de alterações impróprias no estado do sistema;
- **Manutenabilidade** - A habilidade de sofrer reparos e modificações;

A propriedade do sistema que engloba todos estes atributos listados acima é chamada de **dependabilidade**. Avaliação da dependabilidade sempre está ligada ao estudo do efeito de erros, faltas e falhas no sistema, uma vez que estes provocam um impacto negativo nos atributos de dependabilidade. Técnicas de prevenção, predição, remoção e tolerância a faltas, por sua vez, contribuem para manter níveis desejados de dependabilidade, sobretudo em sistemas críticos (Avizienis et al. 2001). Uma **falha** (*fault*,

na terminologia em inglês) é definida como a falha de um componente, subsistema ou sistema que interage com o sistema em questão (Maciel et al. 2012). Um **erro** é definido como um estado que pode levar a ocorrência de uma falha. Um **defeito** (*failure*, na terminologia em inglês) representa o desvio do funcionamento correto de um sistema.

A disponibilidade pode ser definida como a quantificação da alternância entre a provisão correta e incorreta do serviço em questão (Nicol et al. 2004). Ou seja, na avaliação de disponibilidade, levamos em consideração os eventos que podem levar à falha, e também dos eventos que correspondem aos esforços de manutenção empreendidos para levar o sistema novamente ao funcionamento correto. A seguir, definimos formalmente o conceito de disponibilidade estacionária (Maciel et al. 2011):

Definição 1. *A disponibilidade estacionária é definida como a razão entre o tempo de funcionamento esperado e a soma dos tempos de funcionamento e falha esperados:*

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

, onde:

- A *é a disponibilidade estacionária do sistema;*
- $E[Uptime]$ *é o tempo de funcionamento esperado do sistema;*
- $E[Downtime]$ *é o tempo de falha esperado do sistema;*

De maneira similar, a indisponibilidade é calculada pelas equações 2.3 e 2.4

$$UA = \frac{E[Downtime]}{E[Uptime] + E[Downtime]} \tag{2.3}$$

$$UA = 1 - A \tag{2.4}$$

Existem duas métricas derivadas que podem ser úteis para visualizar os efeitos da indisponibilidade sobre o sistema. O ***downtime* anual** representa o número esperado de horas que o sistema estará indisponível no intervalo de um ano, e é calculado pela fórmula da Equação 2.5.

$$Downtime \text{ anual} = UA \times 8760h \tag{2.5}$$

O **número de noves** corresponde a contagem dos algarismos consecutivos iguais a 9, após a vírgula, e é calculado a partir da Equação 2.6 (Marwah et al. 2010).

$$\text{Número de noves} = log_{10}(1-A) \tag{2.6}$$

O número de noves dá um indicativo do nível de disponibilidade que o sistema se encontra. Quanto maior o número de noves, menor a indisponibilidade e *downtime* anual e, provavelmente, mais complexos e custosos os mecanismos de redundância implementados 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. Uma representação visual destes valores é mostrada na Figura 2.2. Esses valores são descritos a seguir.

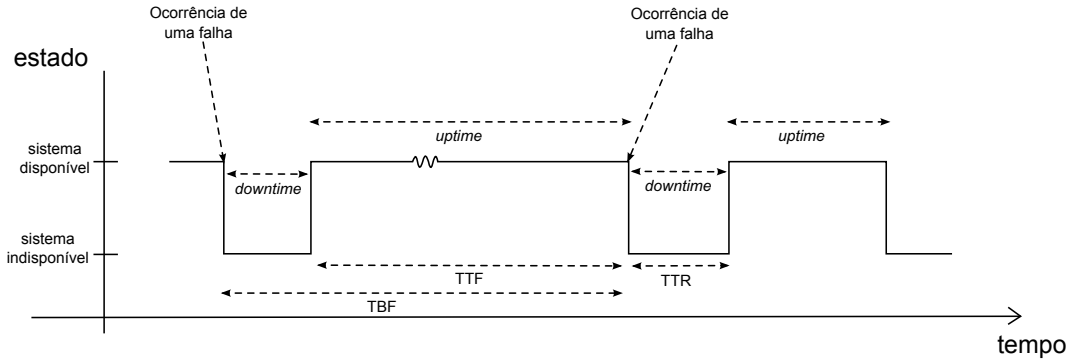


Figura 2.2: *Uptime* e *downtime* (adaptada de (Maciel et al. 2012))

- **Tempo médio para a falha** - *Mean Time To Failure* (MTTF) – É o tempo médio para a ocorrência de falhas do sistema. Corresponde ao valor médio para a métrica TTF da Figura 2.2;
- **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 2.2;
- **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 2.2.

A equação para o cálculo da disponibilidade pode ser escrita em função do MTTF e do MTTR, conforme apresentado na Equação 2.7

$$A = \frac{MTTF}{MTTF + MTTR} \tag{2.7}$$

Quando o $MTTF$ é muito maior que o $MTTR$, a disponibilidade pode ser avaliada de acordo com a Equação 2.8.

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

2.3 Técnicas de Modelagem de Dependabilidade

Nesta seção, introduzimos as técnicas de modelagem de dependabilidade que foram utilizadas neste trabalho. Modelos de dependabilidade são classificados em duas categorias: **modelos combinatoriais** e **modelos baseados em espaço de estados** (Maciel et al. 2012). Os modelos da primeira categoria representam as condições sob as quais um sistema pode se encontrar em falha, ou estar operacional, em termos de relações estruturais entre seus componentes. Modelos baseados em espaços de estados representam o comportamento do sistema através de estados e eventos que provocam transições entre os estados. Da primeira categoria, os modelos mais representativos são as **Árvores de Falhas** e os **Diagramas de Blocos de Confiabilidade**. A segunda categoria tem como principais formalismos as **cadeias de Markov de tempo contínuo** e as **redes de Petri estocásticas** .

Existem alguns prós e contras em cada uma das categorias de modelos (Malhotra 1994). Modelos combinatoriais são mais simples e fáceis de criar, mas devemos assumir que os componentes são estocasticamente independentes. 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, mas eles possuem mais poder de expressividade do que modelos combinatoriais. Com modelos baseado em estados, podemos representar cenários mais complexos, como mecanismos de redundância *cold-standby* ou *warm-standby* (Logothetis & Trivedi 1995).

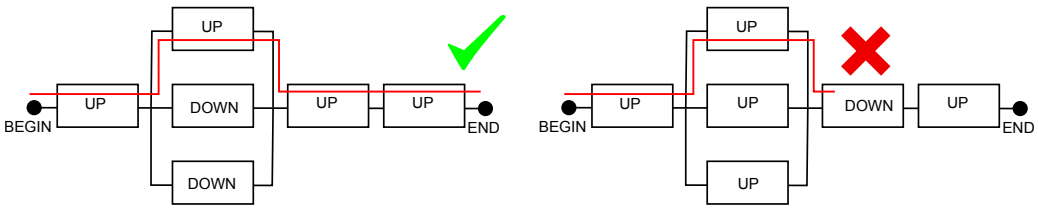
A combinação de ambos os tipos de modelo também é possível, permitindo obter o melhor dos dois mundos, através da **modelagem hierárquica** (Malhotra & Trivedi 1993, Kim et al. 2010). Diferentes modelos, do mesmo tipo ou de tipos distintos, podem ser combinados em diferentes níveis de compreensão, levando a modelos maiores hierárquicos (Malhotra & Trivedi 1993). Geralmente, um modelo de nível superior representa o sistema baseado em seus subsistemas (grupos de componentes relacionados), enquanto que modelos de nível mais baixo descrevem o comportamento detalhado de cada subsistema. Modelos hierárquicos heterogêneos têm sido usados para lidar com a complexidade de vários tipos de sistemas, como por exemplo: redes de sensores (Kim

et al. 2010), redes de telecomunicação (Trivedi et al. 2006) e nuvens privadas (Dantas et al. 2012*b*).

2.3.1 Diagramas de Bloco de Confiabilidade

Diagrama de Blocos de Confiabilidade (*Realibility Block Diagram* - RBD) é um tipo de modelo combinatorial proposto inicialmente para analisar a confiabilidade de sistemas baseados nas relações de seus componentes. Posteriormente, este formalismo foi estendido para a análise de disponibilidade e manutenibilidade. Nesta seção, atentaremos para o uso de RBDs para o cálculo de disponibilidade, uma vez que este é o foco deste trabalho.

Um RBD é formado pelos seguintes componentes: vértice de origem, vértice de destino, componentes do sistema (representados por blocos), e arcos conectando os blocos e os vértices. Um sistema modelado em RBD está disponível caso haja algum caminho do vértice de origem até o vértice de destino, no qual componentes indisponíveis representam uma interrupção em um segmento do caminho. Na Figura 2.3 ilustramos este fato. A Figura 2.3a mostra que o sistema permanece disponível mesmo com a falha de dois componentes, uma vez que há um caminho contínuo dos vértices de início e fim do RBD. Na Figura 2.3b a falha do componente é crítica para o sistema, uma vez que ele impede que haja tal caminho no RBD.

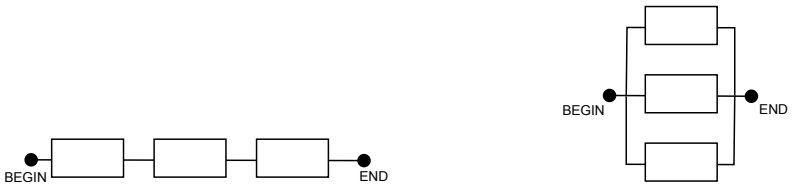


(a) Sistema disponível (b) Sistema indisponível

Figura 2.3: Exemplo de disponibilidade em RBD

Há duas formas principais de arranjar os blocos: em série (Figura 2.4a) e em paralelo (Figura 2.4b). Em um conjunto de componentes em série, todos os componentes deverão estar operacionais para que conjunto esteja disponível. Em conjunto de componentes em paralelo, por sua vez, é necessário que pelo menos um esteja disponível. Outra estrutura típica é a *k-out-of-n*, que é formada por n componentes, e necessita do funcionamento de pelo menos k componentes para estar operacional.

Para avaliar a disponibilidade através de um RBD, introduzimos antes o conceito de *função estrutural* de um sistema. Considere que um sistema S é formado por n componentes: $c_1, c_2, c_3, \ldots c_n$, e este sistema pode estar em falha ou operacional, de



(a) Componentes em série (b) Componentes em paralelo

Figura 2.4: Agrupamentos de componentes em um RBD

acordo com o estado de seus componentes. O estado de um componente c_i é dado pela variável aleatória x_i, tal que:

$$x_i = \begin{cases} 0 & \text{, se o componente falhou} \\ 1 & \text{, se o componente está operacional} \end{cases}$$

A função estrutural de S é definida por $\phi(\mathbf{x})$, onde $x = (x_1, x_2, x_3, \ldots, x_n)$. Da mesma forma que seus componentes, ela pode assumir os valores 0 ou 1.

A **disponibilidade estacionária** de um conjunto de n componentes independentes em série: $b_1, b_2, b_3, \ldots, b_n$ é determinada por:

$$A_S = P\{\phi(x = 1)\} = \prod_{i=1}^{n} P\{x_i = 1\} = \prod_{i=1}^{n} A_i,$$

onde A_i é a disponibilidade estacionária do bloco b_i.

Para uma estrutura em paralelo de n componentes independentes, a disponibilidade estacionária é:

$$A_P = P\{\phi(x = 1)\} = 1 - \prod_{i=1}^{n} UA_i = 1 - \prod_{i=1}^{n} (1 - A_i),$$

onde A_i é a disponibilidade estacionária e UA_i é a indisponibilidade estacionária do componente b_i.

Fazendo aplicações sucessivas destas fórmulas, podemos encontrar a disponibilidade estacionária para um RBD mais complexo formado por diferentes combinações das duas estruturas de composição de blocos. Como exemplo, considere o RBD da Figura 2.5. Para avaliá-lo, damos o primeiro passo que é avaliar a estrutura mais aninhada: a composição em série dos componentes $b2$ e $b4$. Agrupamos estes componentes em um bloco, que possui disponibilidade igual a:

$$A_{b2b4} = A_2 \times A_4$$

Agora, o próximo passo é avaliar a disponibilidade da estrutura em paralelo formada

pelos componentes $b3$ e $b2b4$ (Figura 2.6a):

$$A_{b3b2b4} = 1 - (1 - A_3) \times (1 - A_2 \times A_4)$$

Finalmente, reduzimos o modelo a dois componentes em série (Figura 2.6b), de forma que a disponibilidade do modelo é dada por:

$$A = A_1 \times [1 - (1 - A_3) \times (1 - A_2 \times A_4)]$$

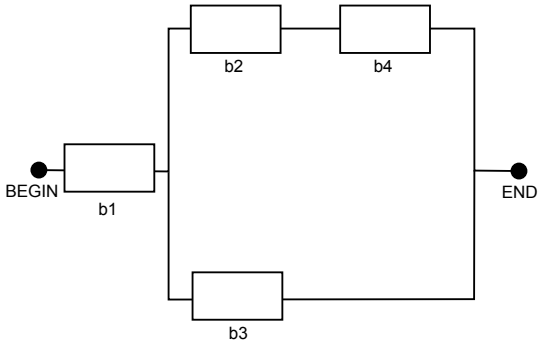
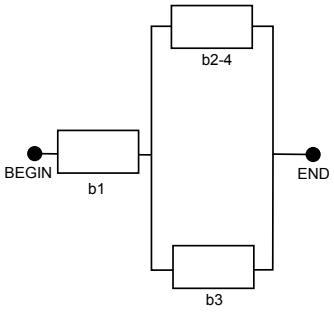
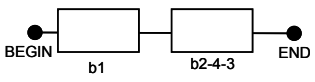


Figura 2.5: Exemplo de RBD com componentes em série e em paralelo



(a) Agrupamento dos componentes $b2$ e $b4$



(b) Agrupamento dos componentes $b3$ e $b2b4$

Figura 2.6: Agrupamentos dos componentes do RBD da Figura 2.5

2.3.2 Cadeias de Markov de Tempo Contínuo

Cadeia de Markov é um modelo matemático proposto por Andrey Markov em 1906 (Markov 1906), que corresponde a um tipo de processo estocástico que possui algumas propriedades especiais. Esta teoria abriu portas para o estudo de processos estocásticos em diversos campos de aplicações, como economia, meteorologia, física, química e

telecomunicações. Na ciência da computação, em particular, este formalismo é bastante conveniente para descrever e analisar propriedades dinâmicas de sistemas computacionais (Bolch et al. 2006). Cadeia de Markov pode ser vista como uma técnica fundamental para a análise de desempenho e dependabilidade, sobre a qual outras técnicas são construídas (Menasce et al. 2004). Como mostraremos na próxima seção, redes de Petri estocásticas podem ser analisadas através da obtenção da cadeia de Markov associada. De forma semelhante, na teoria de redes de filas, uma fila pode ser reduzida a uma cadeia de Markov e analisada e, além disso, alguns teoremas importantes desta teoria são provados através de modelos markovianos (Kleinrock 1975).

Para definir o conceito de cadeia de Markov de tempo contínuo, inicialmente apresentaremos a definição de processo estocástico:

Definição 2. *Um processo estocástico é definido por uma família de variáveis aleatórias $X_t : t \in T$, onde cada variável aleatória X_t é indexada por um parâmetro t e o conjunto de todos os valores possíveis de X_t representa o espaço de estados do processo estocástico.*

Geralmente denominamos o parâmetro t como **parâmetro de tempo**, caso ele satisfaça a condição: $T \subseteq \mathbb{R}^+ = [0, \infty)$. Se o conjunto T é discreto, o processo é classificado como de **tempo discreto**, caso contrário, é de **tempo contínuo**. Da mesma forma, o espaço de estados S pode ser discreto ou contínuo, dividindo processos estocásticos em dois grupos: de **estado discreto** e de **estado contínuo**. Um processo estocástico de espaço discreto pode ser chamado de **cadeia**.

Um processo estocástico de estado discreto é chamado de **processo Markoviano**, caso satisfaça a seguinte propriedade:

Definição 3. *Um processo estocástico $X_t : t \in T$ constitui um processo Markoviano se para todo $0 = t_0 < t_1 < \ldots < t_n < t_{n+1}$ e para todo $s_i \in S$, a probabilidade $P(X_{t_{n+1}} \leq s_{n+1})$ dependa apenas do último valor de X_{t_n}, e não dos valores anteriores $X_{t_0}, X_{t_1}, \ldots, X_{t_{n-1}}$. Ou seja:*

$$P(X_{t_{n+1}} \leq s_{n+1} \mid X_{t_n} = s_n, X_{t_{n-1}} = s_{n-1}, \ldots, X_{t_0} = s_0) =$$
$$P(X_{t_{n+1}} \leq s_{n+1} \mid X_{t_n} = s_n)$$

Em outras palavras, esta propriedade define que, uma vez que um sistema esteja em um determinado estado, a transição para o próximo estado depende apenas do estado atual onde ele se encontra. O histórico de estados que ele esteve anteriormente não importa e não é lembrado, por isso esta propriedade é também chamada de *memoryless property*

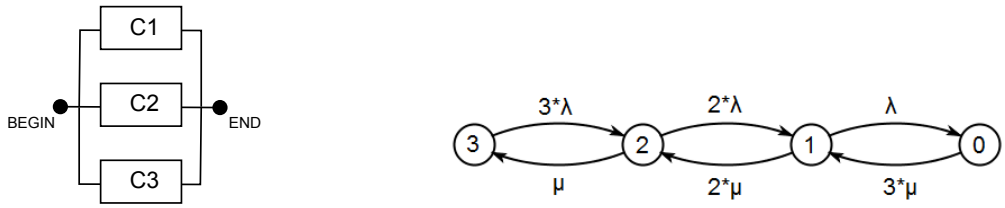
(sem memória). Da mesma forma que os processos estocásticos em geral, dividimos as cadeias de Markov em duas classes, de acordo com o parâmetro de tempo: DTMC (*Discrete Time Markov Chain*), para tempo discreto e CTMC (*Continous Time Markov Chain*), para tempo contínuo. A principal diferença entre essas duas classes é que nas CTMCs, as transições entre os estados podem ocorrer em qualquer instante do tempo, enquanto que nas DTMC, as transições só podem ser feitas em pontos discretos no tempo.

Uma cadeia de Markov pode ser representada como um diagrama de estado, onde os vértices representam os estados, e os arcos do diagrama representam as possíveis transições entre os estados. Os pesos das transições possuem um significado que varia caso a cadeia seja de tempo discreto ou contínuo. Em uma DTMC, o peso de um arco de um estado s_i para s_j corresponde a probabilidade de que, uma vez que o sistema se encontra no estado s_i, aconteça uma mudança para o estado s_j no próximo intervalo de tempo. Consequentemente, esse valor deve ser menor que 1, e a soma de todos os pesos de arcos que saem de um estado s_i também não deve exceder 1.

Nas CTMCs, o peso do arco corresponde a **taxa** de migração de um estado s_i para o estado s_j. O inverso deste valor corresponde ao tempo médio de permanência no estado s_i. A propriedade de não guardar memória das cadeias de Markov implica que este tempo deve ser conduzido por uma distribuição com esta mesma propriedade (Bolch et al. 1998). A única distribuição de tempo contínuo a apresentar essa propriedade é a **exponencial**. Para as DMTCs, a distribuição discreta com essa propriedade é a **geométrica**.

Para ilustrar o método de análise e a representação gráfica de uma CTMC, apresentamos o seguinte exemplo. Considere um sistema cujo modo de operação é representado pelo RBD da Figura 2.7a, que é formado por três componentes em paralelo. Este sistema pode ser representado pela cadeia de Markov da Figura 2.7b. Esta cadeia é formada por quatro estados, e o rótulo de cada estado corresponde ao número de componentes em funcionamento. Desta forma, o sistema está indisponível sempre que entra no estado "0", estando disponível nos demais estados. Considere como λ, a taxa de falha de um componente, e μ, a taxa de reparo. Observe na figura, que do estado "3" para o estado "2" a taxa é de $3 * \lambda$, uma vez que há 3 componentes funcionais sujeitos à falha. As outras taxas são ajustadas de forma semelhante.

Cadeias de markov possuem uma representação em forma de matriz, denominada **matriz de taxa de transição** Q. Um elemento q_{ij} da matriz corresponde à taxa de transição do estado i para o estado j. Elementos da diagonal q_{ii} são definidos como o somatório dos elementos da mesma linha, multiplicado por -1. Para o exemplo da Figura 2.7b, temos:



(a) Sistema representado como RBD (b) Sistema representado como CTMC

Figura 2.7: Exemplo de CTMC para análise

$$Q = \begin{pmatrix} -3*\lambda & 3*\lambda & 0 & 0 \\ mu & -mu-2*\lambda & 2*lambda & 0 \\ 0 & 2*\mu & -2*\mu-\lambda & \lambda \\ 0 & 0 & 3*\mu & -3*\mu \end{pmatrix}$$

A **análise estacionária** de uma cadeia de Markov consiste em encontrar a probabilidade de encontrar o sistema em um determinado estado, considerando um longo tempo de execução. Estas probabilidades são independentes do estado inicial do sistema e são representadas pelo vetor $\pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$, onde π_i é a probabilidade estacionária para o estado i. Uma vez que a cadeia de Markov considerada é uma cadeia **ergódica**, podemos encontrar as probabilidades estacionárias através do sistema linear formado pelas Equações 2.9 e 2.10 (Bolch et al. 2006). Uma cadeia de Markov é dita ergódica, se é possível alcançar qualquer estado a partir de qualquer outro estado, em um número n finito de passos. A Equação 2.9 produz as **equações de balanço** da cadeia de Markov e significa que a soma do fluxo de entrada em um determinado estado com o fluxo de saída deve ser igual a zero. A Equação 2.10 diz que as probabilidades estacionárias são mutuamente exclusivas e exaustivas, portanto, a soma de todas elas deve ser igual a 1.

$$\pi Q = 0 \tag{2.9}$$

$$\sum_{i=1}^{n} \pi_i = 1 \tag{2.10}$$

Para o exemplo da Figura 2.7b, as Equações 2.9 e 2.10 produzem o sistema linear 2.11.

$$\begin{aligned} -3\lambda\pi_0 + \mu\pi_1 &= 0 \\ 3\lambda\pi_0 + (-\mu - 2*\lambda)\pi_1 + 2*\mu\pi_2 &= 0 \\ 2\lambda\pi_1 + (-2*\mu - \lambda)pi2 + 3*\mu\pi_3 &= 0 \\ \lambda\pi_2 + -3*\mu\pi_3 &= 0 \\ \pi_0 + \pi_1 + \pi_2 + \pi_3 &= 1 \end{aligned} \tag{2.11}$$

Considere que um componente do exemplo possui tempo médio de falha (exponencial) de 1200 horas, e um tempo médio de reparo de 24 horas. Desta forma, os parâmetros do modelo serão $\lambda = 1/1200$ e $\mu = 1/24$. Fixando estes parâmetros e resolvendo o sistema linear obtido anteriormente, obtemos as probabilidades estacionárias da CTMC. A probabilidade estacionária π_3 equivale à probabilidade do sistema ser encontrado sem nenhum componente em funcionamento. Desta forma, a disponibilidade do sistema é igual a:

$$A = 1 - \pi_3 = 0,99999246\ ,$$

que corresponde ao mesmo valor obtido ao resolver o modelo em RBD da Figura 2.7b.

2.3.3 Redes de Petri

O termo Redes de Petri se refere a uma família de formalismos matemáticos adequados para a representação e análise de propriedades de sistemas concorrentes, que apresentam características como escolha, execução paralela de atividades e compartilhamento de recursos. O conceito foi inicialmente proposto em 1962 por Carl Adam Petri em sua tese de doutorado e, desde então, vem se expandindo com inúmeras extensões, algoritmos de verificação e aplicações nas mais diversas áreas do conhecimento.

O tipo mais básico de redes de Petri é denominado *Place/Transition*. Também pode receber o termo rede de Petri elementar (Wolfgang Reisig 2013), ou simplesmente rede de Petri (quando o tipo específico é omitido, podemos considerar que a rede é do tipo *Place/Transition* (Reisig & Rozenberg 1998)). Na Figura 2.8 mostramos algumas das principais extensões (a lista não é completa) das redes de Petri. Algumas extensões adicionam a noção de tempo às redes, e outras adicionam mecanismos de alto nível, como *tokens* coloridos, objetos ou a noção de hierarquia. Nesta seção, detalhamos as redes *Place/Transition* e posteriormente as Redes de Petri Estocásticas e Generalizadas, que foi a extensão utilizada neste trabalho.

Uma rede de Petri P/T é representada por um grafo direcionado bipartido formado por

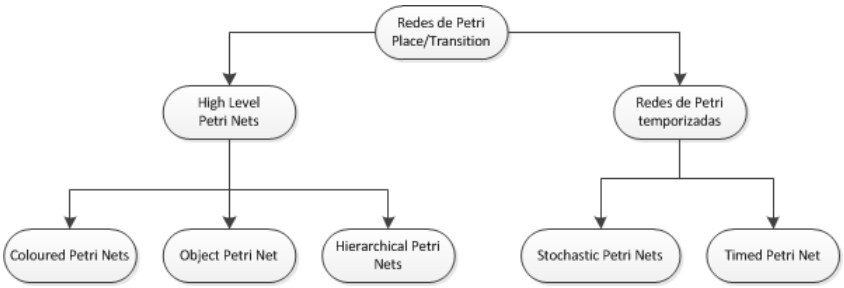


Figura 2.8: Tipos de redes de petri

dois conjuntos de vértices: lugares e transições. Por ser um grafo bipartido, não temos arcos ligando vértices do mesmo tipo: lugar com lugar ou transição com transição. Um exemplo de rede de Petri pode ser visualizado na Figura 2.9. Este modelo representa uma linha de produção de um determinado produto. Um produto é fabricado por uma máquina, utilizando três parafusos e três porcas e, após ser montado, é colocado em um depósito. Os componentes da rede de Petri são detalhados a seguir:

- **Lugar** - São os vértices em forma de círculo. Representa um componente passivo da rede, e geralmente é usado para armazenar ou acumular coisas (porcas, parafusos, produtos), ou indicar um estado do modelo, como a disponibilidade de um determinado recurso (a máquina, por exemplo);
- **Token** - São elementos que residem nos lugares, representados como pontos pretos, e servem para representar o estado do sistema: quantos parafusos/porcas temos no estoque, a disponibilidade da máquina, quantos produtos já foram produzidos, etc.
- **Transição** - São os vértices em forma de retângulo. São os componentes ativos de uma rede de Petri, e representam ações que causam a mudança no estado do sistema, como a montagem de um produto, o armazenamento de um produto em depósito, etc. O disparo de uma ação causa mudança na marcação de uma rede de Petri, de acordo com os arcos que conectam a transição aos lugares.
- **Arco** - Determinam as relações entre lugares e transições. Um arco que tem como origem um lugar e destino uma transição é chamado *arco de entrada*, enquanto que um arco com origem em uma transição e destino em um lugar, é denominado *arco de saída*.

Uma transição t possui dois conjuntos associados. O *pre-set* de t, representado pelo símbolo ${}^\bullet t$, representa o conjunto de lugares que estão conectados a t por um arco de

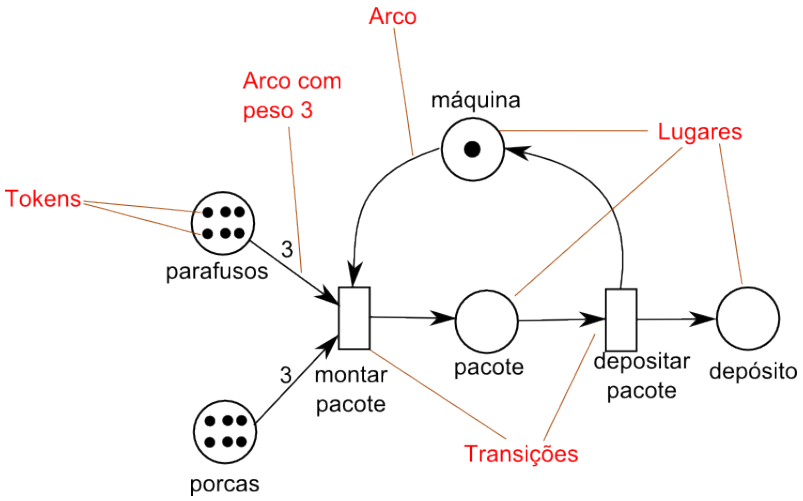


Figura 2.9: Rede de petri

entrada. O *post-set* de t , representado por $t^\bullet$, representa o conjunto de lugares conectados a t por arcos de saída. Na Figura 2.10, destacamos o *pre-set* e o *post-set* da transição "montar pacote". Uma transição t está **habilitada**, se cada lugar do seu *pre-set* possuir uma quantidade de *tokens* superior ao peso do arco que o conecta à transição. O **disparo** de uma transição é o evento que provoca a mudança da marcação da rede de Petri, isto é, uma mudança no estado do sistema. Uma transição pode disparar apenas se estiver habilitada. O disparo de uma transição provoca a retirada de *tokens* dos lugares do *pre-set*, e a criação de novos *tokens* nos lugares do *post-set* da transição. O número de *tokens* retirados e colocados pela transição é determinado pelo peso do arco. Na Figura 2.11 mostramos o estado da rede antes e após o disparo da transição "montar pacote".

Tendo explicado uma rede de Petri através dos seus mecanismos visuais, introduzimos a definição formal de uma rede de Petri:

Definição 4. *Uma rede de Petri* Place/Transition *é um grafo direcionado bipartido representado por uma tupla* $R = (P, T, F, W, m_0)$*, na qual:*

- $P \cup T$ *são os vértices do grafo.* P *é o conjunto dos lugares e* T *é o conjunto das transições da rede*
- P *e* T *são conjuntos disjuntos, isto é,* $P \cap T = \varnothing$
- F *é o conjunto das arestas de* R*, onde* $F \subseteq A = (P \times T) \cup (T \times P)$
- $W : A \to \mathbb{N}$ *é o conjunto que representa os pesos dos arcos*

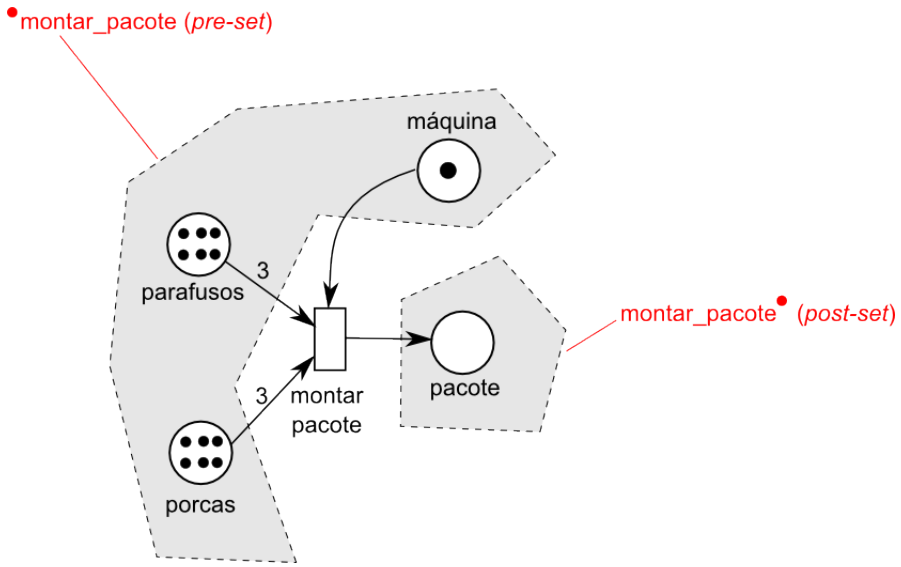


Figura 2.10: *Pre-set* e *post-set*

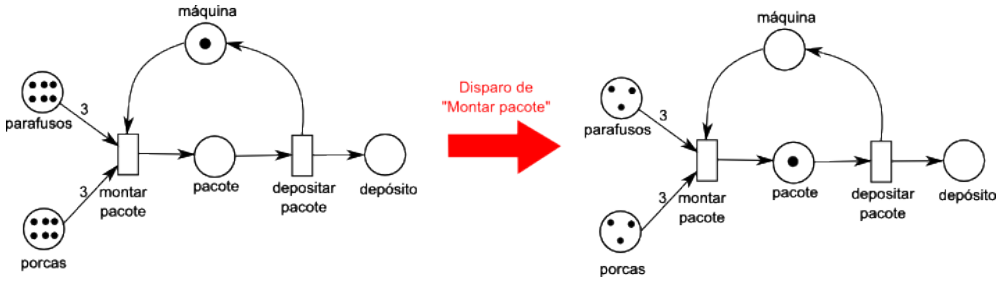


Figura 2.11: Exemplo de disparo de uma transição

- m_i *representa uma marcação da rede de Petri e é determinada por uma função* $m_i : P \to \mathbb{N}$. m_0 *representa a marcação inicial da rede de Petri*

Chamamos a tupla formada por (P,T,F,W) como a *estrutura da rede de Petri*, enquanto que a tupla $R = (P,T,F,W,m_0)$ é denominada *rede de Petri marcada*.

O **grafo de alcançabilidade** de uma rede de Petri é um grafo direcionado cujo conjunto de vértices é formado pelo conjunto de marcações alcançáveis, e o conjunto de arestas é formado pelas transições que provocaram a mudança das marcações. Com esse grafo, podemos verificar muitas propriedades importantes das redes de Petri e, como mostramos na seção seguinte, obter a CTMC associada a uma rede de Petri estocástica. Na Figura 2.12, mostramos o grafo de alcançabilidade da rede de Petri da linha de produção. A marcação está representada entre chaves no rótulo do estado, e cada número corresponde ao número de *tokens* nos lugares: "parafusos", "porcas", "pacote", "depósito" e "máquina", respectivamente.

Figura 2.12: Grafo de alcançabilidade

2.3.3.1 Redes de Petri Estocásticas Generalizadas

Rede de Petri Estocástica Generalizada (GSPN - *Generalized Stochastic Petri Nets*) é uma extensão das redes *Place/Transition*, onde as transições podem ser temporizadas ou imediatas (Chiola et al. 1993). Diferente das *Timed Petri Nets* (Zuberek 1991) e *Time Petri Nets* (Berthomieu & Diaz 1991), que possuem tempos de disparos determinísticos, os tempos de disparos das transições temporizadas de uma GSPN são aleatórios e exponencialmente distribuídos. As transições imediatas, como o nome implica, são disparadas automaticamente quando se tornam habilitadas, sem nenhum atraso. Uma propriedade interessante da GSPN é que, assumindo que ela possua um grafo de marcação finito e possua as propriedades de reversibilidade[2] e ausência de *deadlock*[3], esta rede possui uma cadeia de Markov de tempo contínuo associada (Ciardo et al. 1994). Desta forma, podemos utilizar GSPN para modelar um sistema complexo que seria difícil de ser modelado com uma cadeia de Markov com um grande número de estados. Com as ferramentas apropriadas, podemos analisar o modelo em GSPN da mesma forma que analisaríamos uma CTMC, obtendo probabilidades estacionárias e transientes de determinadas marcações da rede, que correspondem aos estados da CTMC embutida.

A seguir, apresentamos alguns conceitos introduzidos pela notação GSPN:

- **Arco inibidor** - Este tipo especial de arco é representado na notação gráfica com um círculo branco na extremidade do arco, em vez da seta que é utilizada nos arcos normais. Um arco inibidor sempre tem como origem um lugar e uma transição como destino, e pode ter um peso associado. Diferente dos arcos normais, este não causa movimentação de *tokens* entre os lugares. Sua função é desabilitar a transição sempre que o lugar associado ao arco tiver um número de *tokens* menor ou igual ao peso do arco;

- **Prioridades associadas às transições imediatas** - Uma vez que transições imediatas sempre disparam automaticamente após estarem habilitadas, é necessário

[2]Uma rede de Petri é reversível se, em qualquer marcação M_k alcançável a partir da marcação M_0, M_0 é alcançável a partir de M_k.

[3]Um *deadlock* corresponde a uma marcação na qual nenhuma transição está habilitada.

algum tipo de mecanismo para resolver conflitos quando duas ou mais transições imediatas estejam habilitadas. Uma forma de resolver isso é atribuindo prioridades às transições. Se várias transições imediatas estiverem habilitadas simultaneamente, a que tiver maior prioridade será a que sofrerá um disparo;

- **Pesos associadas às transições imediatas** - Quando duas ou mais transições de mesma prioridade estão habilitadas ao mesmo tempo, a escolha da transição que irá disparar é feita não-deterministicamente. Nesta situação, a probabilidade de uma transição disparar é igual a:

$$Prob = \frac{\text{Peso da transição}}{\text{Somatório de todos os pesos das transições imediatas habilitadas}}$$

Por exemplo, se tivermos três transições imediatas de mesma prioridade habilitadas, duas com peso 1, e uma com peso 2, as probabilidades de disparo serão de 0.25, 0.25 e 0.5, respectivamente;

- **Semântica de servidor** - Uma transição temporizada pode permitir múltiplos disparos em paralelo. Este comportamento é determinado pela semântica de servidor da transição. Uma transição do tipo *single server* não possui nenhum tipo de concorrência, e todos os disparos são feitos sequencialmente. Transições do tipo *multiple server* (ou *k server*) permitem que sejam efetuados k disparos em paralelo. Uma transição do tipo *infinite server* permite tantos disparos em paralelo quanto a marcação da rede permitir. Para ilustrar este conceito, considere a rede modificada da Figura 2.9, ilustrada na Figura 2.13. Caso a transição "montar pacote" seja single server, a máquina produzirá pacotes sequencialmente. Caso se utilize a semântica *multiple server* com $k = 3$, o modelo permitirá que três pares de porca e parafuso sejam processados em paralelo. Os outros três pares que sobraram irão aguardar término do processamento dos pacotes em produção. Na semântica *infinite server*, os seis pares de porcas e parafusos serão processados em paralelo.

Para ilustrar estes conceitos, apresentamos na Figura 2.14, um exemplo de uma GSPN. Este exemplo de GSPN, extraído de German (1996), representa uma fila M/M/1/K. A transição *gerar* cria os *tokens* que correspondem às requisições de serviço. Cada *token* criado por esta transição é depositado no lugar *gerado* e, a partir daí, uma escolha é feita. O *token* pode ser enfileirado para ser processado pelo servidor, caso haja espaço em fila, ou pode ser descartado pela transição imediata *perda*, quando todos os k lugares da fila

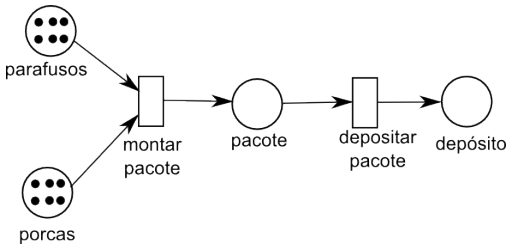


Figura 2.13: Exemplo de semântica de servidor

estão ocupados. O lugar *livres* controla o disparo das transições imediatas que fazem este controle. A transição *perda* só estará ativada quando não houver mais *tokens* em *livres*, devido a presença do arco inibidor. E a transição *enfileira* só estará habilitada se houver um ou mais *tokens* em *livres*, devido ao arco de entrada. A transição *servir* representa o processamento das requisições pelo servidor. Uma vez que a fila é M/M/1/K, então só há um servidor para atender todas as requisições. Desta forma, a semântica de servidor da transição *servir* é *single server*.

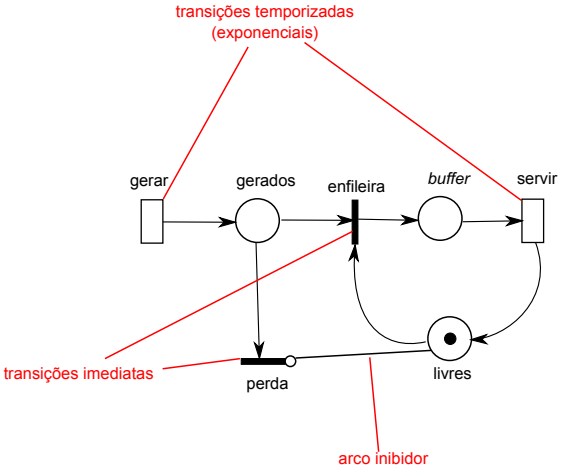


Figura 2.14: Rede de petri estocástica (German 1996)

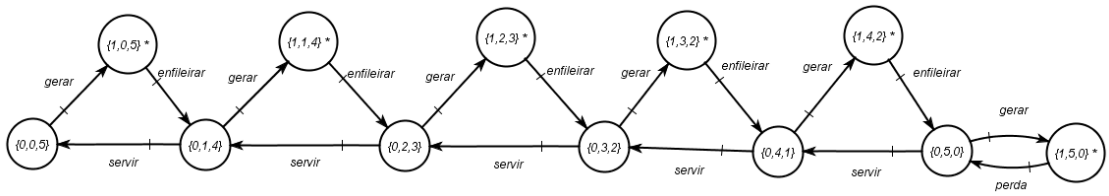
Denominamos de SPN (*Stochastic Petri Net*) um tipo mais simples de GSPN, que não possui transições imediatas e cujas transições seguem a semântica de *single server*. Uma SPN possui uma CTMC associada, que é obtida da seguinte forma (Kartson et al. 1994):

1. O espaço de estados da CTMC equivale ao grafo de alcançabilidade da rede de Petri associada à SPN, no qual a marcação M_i do grafo de alcançabilidade equivale ao estado s_i da CTMC;

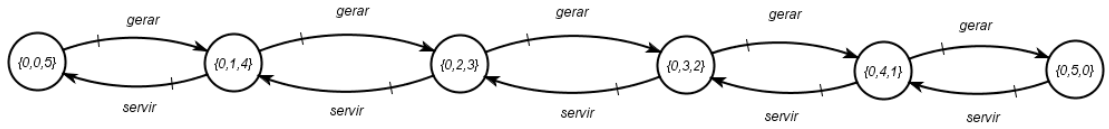
2. A taxa de transição de um estado s_i para um estado s_j é igual à soma de todas as taxas de disparo das transições que estão habilitadas em M_i, que cujos disparos

produzem a marcação M_j.

Devido à presença de transições imediatas, o grafo de alcançabilidade de uma GSPN produz dois tipos distintos de marcações. Marcações que possuem apenas transições imediatas habilitadas são denominadas "marcações *vanishing*", enquanto que marcações que possuem apenas transições temporizadas habilitadas são denominadas "marcações tangíveis". Em uma GSPN, temos apenas estes dois tipos de marcações, uma vez que é impossível uma marcação possuir os dois tipos de transições habilitadas ao mesmo tempo. O tempo médio de permanência em transições *vanishing* é igual a zero, desta forma, tais estados não são incluídos na CTMC associada a uma GSPN. Para obter esta CTMC, é preciso remover os estados *vanishing* e ajustar as transições e suas respectivas taxas. Na Figura 2.15a mostramos o grafo de alcançabilidade da GSPN da Figura 2.14. Os estados *vanishing* são indicados com um asterisco. A CTMC obtida após a remoção destes estados é mostrada na Figura 2.15b.



(a) Grafo de alcançabilidade com estados *vanishing*



(b) Cadeia de Markov embutida da rede de Petri

Figura 2.15: Grafo de alcançabilidade e CTMC embutida de uma rede de Petri

2.4 Ferramentas de Modelagem Analítica

Nesta seção, fazemos uma descrição breve das ferramentas utilizadas para a criação e avaliação dos modelos deste trabalho: SHARPE, TimeNET e Mercury. Concluímos esta seção com uma discussão comparativa sobre essas três ferramentas.

2.4.1 SHARPE

A ferramenta SHARPE (*Symbolic Hierarchical Automated Reliability and Performance Evaluator*) (Sahner et al. 1996) é uma ferramenta voltada para a análise de confiabilidade, disponibilidade, desempenho e performabilidade. Dentre as ferramentas de modelagem consideradas neste trabalho, esta é a que suporta o maior número de formalismos, que são listados a seguir:

- Árvore de Falha
- Diagramas de Bloco de Confiabilidade (RBD)
- Grafo de Confiabilidade
- Cadeia de Markov
- Modelo MRGP
- Rede de Petri Estocástica Generalizada (GSPN)
- Rede de Petri Estocástica de Recompensa
- *Product Form Queueing Network* (PFQN)
- *Multi Chain* PFQN
- Grafo

A partir do nome da ferramenta, podemos destacar duas características importantes que ela apresenta. A primeira é que modelos podem ser parametrizados por variáveis **simbólicas**. As entradas da matriz geradora de modelos markovianos também são armazenadas de forma simbólica, ao passo que a maioria das ferramentas armazenam as entradas sob forma numérica (Matos 2011). Isso confere grande flexibilidade na análise de modelos e facilita a análise de sensibilidade de parâmetros. A segunda característica é o suporte à **modelagem hierárquica**, que permite o usuário combinar modelos de formalismos distintos, de forma que modelos de níveis inferiores atuem como parâmetros de entrada para modelos de níveis superiores.

A ferramenta SHARPE pode ser obtida em (Trivedi 2010).

2.4.2 TimeNET

A ferramenta TimeNET (Zimmermann et al. 2000) suporta a criação e análise de modelos de duas categorias de redes de Petri: redes de Petri estocásticas e redes de Petri estocásticas coloridas. As redes de Petri estocásticas simples (não coloridas) podem ser avaliadas por métodos numéricos de análise transiente e estacionária e por simulação, caso o espaço de estados seja muito grande. Redes de Petri estocásticas coloridas só podem ser avaliadas por simulação. Redes de Petri coloridas podem ter *tokens* associados a um tipo de dados (inteiro, real, etc.) e o tempo médio de disparo das transições podem seguir outras distribuições além da exponencial.

Os resultados da avaliação de uma rede de Petri no TimeNET são obtidos através da criação de **métricas**. Tais métricas seguem uma sintaxe particular, sendo representadas como uma expressão que pode conter números, operadores algébricos e as seguintes medidas básicas:

- $P\{$<*expressão*>$\}$, que corresponde a probabilidade (transiente ou estacionária), de que a rede se encontre em uma determinada marcação que satisfaz a expressão booleana entre as chaves.
- $E\{$#<*nome do lugar*>$\}$, que corresponde ao número médio de *tokens* em um determinado lugar.

As expressões utilizam a sintaxe # $<$ *nome do lugar* $>$, para referenciar um lugar da rede de Petri em particular. Para ilustrar o uso das medidas básicas, considere a GSPN da Figura 2.14. Podemos calcular o número médio de requisições enfileiradas através da expressão: $E\{\#\text{buffer}\}$. O *throughput* do servidor é calculado pelo produto da taxa da transição *servir* pela a soma das probabilidades estacionárias de todas as marcações que habilitam esta transição. Suponha que a transição *servir* possua tempo médio de disparo igual a 2. A métrica para a vazão da transição, na sintaxe do TimeNET é dada pela expressão:

$$(1/2) * P\{\#\text{buffer} > 0\}$$

A expressão $P\{\#\text{buffer} > 0\}$ corresponde à soma das probabilidades estacionárias de todas as marcações cujo número de *tokens* no lugar *buffer* é maior que zero. Podemos combinar várias condições associadas a lugares com os operadores lógicos *AND* e *OR*, e definir a precedência das operações com o uso de parênteses.

A ferramenta TimeNET pode ser obtida em (Zimmermann 2013).

2.4.3 Mercury

A ferramenta Mercury (Silva et al. 2013) é o *kernel* de outra ferramenta denominada ASTRO (Callou et al. 2013). A ferramenta ASTRO é um ambiente integrado para a avaliação de sustentabilidade, custo e disponibilidade de ambientes de *data center*. Esta ferramenta permite que usuários modelem sistemas de TI, sistemas energéticos e sistemas de resfriamento, sem a necessidade de conhecer os formalismos matemáticos que serão empregados para realizar a análise das propriedades desses sistemas.

O Mercury, por sua vez, além de ser o *kernel* de modelagem e avaliação dos modelos de alto nível criados pelo ASTRO, também é uma ferramenta para a criação e análise de modelos de RBD, cadeias de Markov, redes de Petri estocásticas e modelos de fluxo energético (EFM). A relação entre as ferramentas ASTRO e Mercury é visualizada na Figura 2.16.

Para a avaliação de redes de Petri, a ferramenta Mercury adota o conceito de métricas e utiliza uma sintaxe semelhante à utilizada pela ferramenta TimeNET.

Consulte (Silva 2013) para obter informações sobre como fazer o *download* da ferramenta.

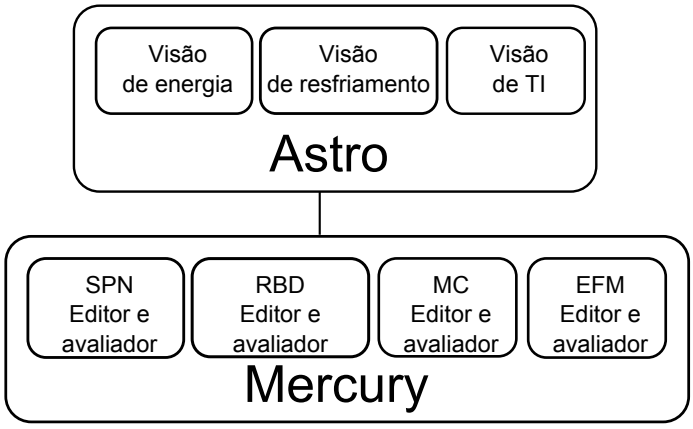


Figura 2.16: Funcionalidades do ambiente ASTRO

2.4.4 Comparativo Entre as Ferramentas

Para a criação e avaliação de cadeias de Markov e RBDs, tanto o SHARPE quanto o Mercury podem ser usados. Já no caso das redes de Petri estocásticas, as três ferramentas podem ser utilizadas. A princípio pode parecer redundante o uso de diferentes ferramentas, quando apenas uma poderia atender as necessidades. Contudo, cada uma delas possui

alguns pontos fortes e fracos que iremos detalhar a seguir. Acreditamos que o uso combinado das três pode promover melhores resultados.

Para a avaliação de RBDs, o SHARPE e o Mercury fornecem o mesmo conjunto básico de operações: cálculo de disponibilidade, confiabilidade, tempo médio até a falha (MTTF) e tempo médio até o reparo (MTTF). O Mercury adicionalmente fornece opções para: análise de *reliability importance*, cálculo de custo embutido e analise de sensibilidade. Além disso, o Mercury também permite obter as funções estruturais e lógicas de um RBD. O SHARPE, por sua vez, possui mecanismos para criar modelos RBD hierárquicos, enquanto que o Mercury ainda não possui esta funcionalidade[4].

De forma semelhante, para CTMCs, as duas ferramentas citadas possuem o mesmo conjunto básico de operações: cálculo das probabilidades estacionárias e transientes e cálculo para o tempo médio de absorção. O SHARPE adicionalmente oferece suporte ao cálculo de taxas de recompensa, enquanto que o Mercury não apresenta essa opção.

Para finalizar a comparação entre o SHARPE e Mercury, ressaltamos que o Mercury tem uma interface mais moderna e amigável que a ferramenta SHARPE. O SHARPE permite a utilização em modo de linha de comando e uma linguagem de *script*, que é uma solução robusta e flexível, mas que implica em uma curva de aprendizado relacionada ao domínio da sintaxe dos comandos e da linguagem.

Diferente das ferramentas SHARPE e Mercury, a TimeNET é voltada exclusivamente para um formalismo - as redes de Petri estocásticas. Como diferencial sobre as outras ferramentas, ela apresenta funcionalidades de verificação de propriedades estruturais da rede de Petri e cálculo do espaço de estados. Ela também permite a representação de transições temporizadas nas quais a taxa de disparo é condicionada a uma expressão lógica. Por último, destacamos que o TimeNET oferece suporte à redes de Petri coloridas, que é um formalismo de nível mais alto que as redes de Petri estocásticas simples.

Para a avaliação de redes de Petri, a ferramenta SHARPE oferece um conjunto pré-definido de operações, em vez de permitir que o usuário construa expressões com os operadores $P\{\ldots\}$ e $E\{\ldots\}$. Embora essa abordagem seja menos flexível, o SHARPE possui um diferencial sobre o TimeNET e o Mercury. Com o SHARPE é possível avaliar o tempo médio até a falha de uma rede de Petri. Isto é, considerando uma rede de Petri que tenha uma marcação alcançável em *deadlock*, o SHARPE é capaz de calcular o tempo médio para a rede atingir esta marcação. Esta funcionalidade é usada para avaliação do modelo de consumo energético apresentado na Seção 4.6.

[4]Embora seja possível realizar modelagem hierárquica com o Mercury, computando os resultados dos modelos de níveis mais baixo de forma separada e inserindo manualmente no modelo de alto nível

Uma funcionalidade exclusiva da ferramenta Mercury em relação às demais, é a aproximação de uma distribuição empírica por uma distribuição *phase-type* (hipoexponencial, hiperexponencial ou *Erlang*) através da técnica de *moment matching* (Watson & Desrochers 1991). 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. Utilizamos esta funcionalidade para a criação dos modelos de disponibilidade e consumo energético apresentados na Seção 4.6.

2.5 Injeção de Falhas

Ao criar experimentos que avaliam atributos de dependabilidade do sistema, é necessária uma coleção de ferramentas que permitam controlar a ocorrência das falhas, uma vez que estas são imprevisíveis, e podem levar um longo tempo para ocorrer. Técnicas de injeção de falhas permitem a observação do comportamento do sistema através de experimentos controlados, nos quais a presença de falhas é induzida explicitamente nos componentes do sistema (Arlat et al. 1990).

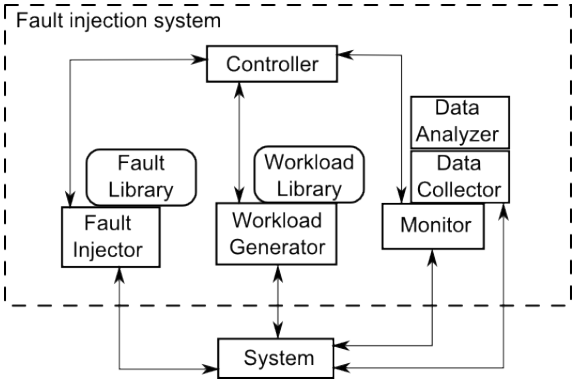


Figura 2.17: Arquitetura para um *testbed* de injeção de falhas

A Figura 2.17 mostra a arquitetura básica de um ambiente de injeção de falhas e monitoramento de atributos de dependabilidade (Hsueh et al. 1997). O *Controller* é o módulo que gerencia os outros componentes, e controla o experimento. O *Fault Injector* atua injetando falhas nos componentes do sistema, a medida que este executa operações enviadas pelo *Workload Generator*. As falhas injetadas são especificadas pelo

módulo *Fault Library*. O *Monitor* observa o comportamento do sistema, e envia os dados observados para o *Data Collector*. A análise dos dados pode ser executada em tempo real ou *offline* pelo *Data Analyzer*. A ferramenta de injeção de falhas criada neste trabalho para a plataforma *Android* segue uma adaptação desta arquitetura básica.

Segundo Ziade et al. (2004), há dois tipos de implementação de um injetor de falhas: injetores de falha em nível de *hardware*, e injetores de falha em nível de *software*. Os injetores de falhas baseados em *hardware* podem ser classificados em duas categorias: com contato e sem contato. Nos injetores de falhas baseados em *hardware* com contato, o injetor é inserido fisicamente no mesmo circuito onde estarão os componentes onde as falhas serão induzidas. Isto pode ser realizado através de mudanças na voltagem ou da corrente do componente. Nos injetores sem contato, não há conexão física entre o injetor e o componente, em vez disso, são utilizadas técnicas de radiação de íons ou interferência eletromagnética para afetar o *chip* alvo.

Da mesma forma que falhas injetadas via *hardware*, as técnicas de injeção de falhas por *software* são divididas em duas categorias: em tempo de compilação ou em tempo de execução (Hsueh et al. 1997). 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. Este método trás como vantagens não ter a necessidade de um *software* adicional para o mecanismo de injeção, e oferece pouca perturbação ao sistema original. Contudo, caso não se tenha acesso ao código fonte do sistema original, não é possível usar esta técnica. 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:

- ***Time-out* -** Esta técnica utiliza um *timer* (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 tal exigência, sendo consideravelmente menos onerosa. Também existe um 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* também 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. Entretanto, há alguns pontos que só são possíveis ser atingidos com os injetores de *hardware*, como circuitos VLSI. 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. De forma resumida, podemos considerar que a injeção de falhas por *software* deverá ser a primeira opção, uma vez que é menos custosa e mais flexível. Porém, caso exista uma necessidade real de injetar falhas que não são possíveis de ser provocadas por *softwares*, ou caso o sistema alvo seja crítico e a perturbação causada pelo *software* injetor seja significativa, devemos considerar a utilização de injetores de falhas em *hardware*.

3

Metodologia de Avaliação de Dependabilidade para *Mobile Cloud Computing*

A metodologia proposta é descrita pela Figura 3.1, que contém um fluxograma com a sequência das atividades realizadas. A primeira etapa é a definição de uma arquitetura básica de *mobile cloud*. Além de definir a arquitetura básica, também estudamos os componentes dessa arquitetura sob o ponto de vista da disponibilidade, isto é, os componentes sob os quais eventos de falha poderão afetar o funcionamento correto do sistema. Em seguida, elaboramos um modelo hierárquico de disponibilidade do sistema formado por RBD e CTMC. O mesmo cenário foi avaliado também através de outras duas técnicas: simulação e experimentos de injeção de falhas. Para realizar o experimento de injeção de falhas, construímos um *testbed* de monitoramento de disponibilidade e injeção de falhas que está descrito na Seção 3.2. A API de Simulação foi construída com base no NS-2 Network Simulator, e está descrita na Seção 3.3.

Em posse dos resultados obtidos através das três técnicas de avaliação para a arquitetura básica (modelagem analítica, simulação e experimento de injeção de falhas), realizamos a validação do modelo criado. Descrevemos esta etapa na Seção 5.1. Tendo validado o modelo, podemos obter um maior nível de confiança nele e, posteriormente, estendê-lo com o objetivo de investigar extensões da arquitetura básica que promovam um aumento na disponibilidade. São concebidos novos modelos para estas extensões e os resultados da análise destes modelos são discutidos a fim de obter *insights* sobre a eficácia das soluções escolhidas sobre a melhoria na disponibilidade do sistema. Estes modelos são apresentados no Capítulo 4. Dos cenários propostos, um é selecionado com o objetivo de estudar o efeito combinado do uso de interfaces *wireless* sobre o consumo

energético e sobre a disponibilidade.

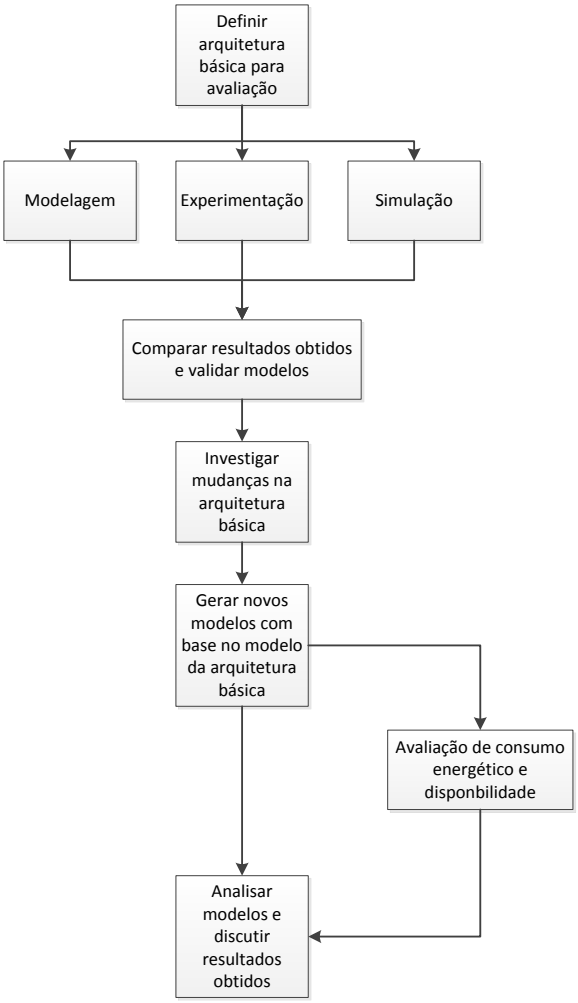


Figura 3.1: Metodologia da dissertação

3.1 Arquitetura Básica

Na Figura 3.2, exibimos uma representação da arquitetura básica. A arquitetura básica é formada por três componentes: o cliente móvel, o serviço em nuvem, e uma conexão *wireless* WLAN entre estes dois. Na seção 4.1 apresentamos o modelo de disponibilidade para esta arquitetura e detalhamos cada componente da arquitetura básica que influencia na disponibilidade do sistema.

Na nossa arquitetura básica, o serviço em nuvem é provido por um servidor *web* e por um banco de dados. Estes dois servidores são implantados em de máquinas virtuais

que executam na nuvem. Cada máquina virtual executa em uma nó de *cluster* diferente da infraestrutura. Os nós de *cluster* são os componentes da nuvem responsáveis por oferecer seus recursos computacionais para os clientes da nuvem através de virtualização. Enfatizamos que a infraestrutura em nuvem é formada por mais componentes além dos dois nós de *cluster* que executam as máquinas virtuais que fornecem o serviço aos clientes móveis. Contudo, a falha nos demais componentes da nuvem: nós de *cluster* adicionais, gerenciador da infraestrutura, gerenciador de armazenamento, etc. **não** provoca impacto na disponibilidade do ponto de vista do cliente móvel. A falha destes poderiam afetar apenas a habilidade de implantar novas máquinas virtuais ou executar tarefas administrativas nos nós de cluster. Enquanto os nós sob os quais as máquinas virtuais que proveem o serviço para os clientes estiverem funcionando corretamente, os clientes estarão aptos a consumir o serviço. Na Seção 4.5, apresentamos uma extensão da arquitetura básica que utiliza as funcionalidades destes componentes adicionais para aumentar a disponibilidade do serviço em nuvem.

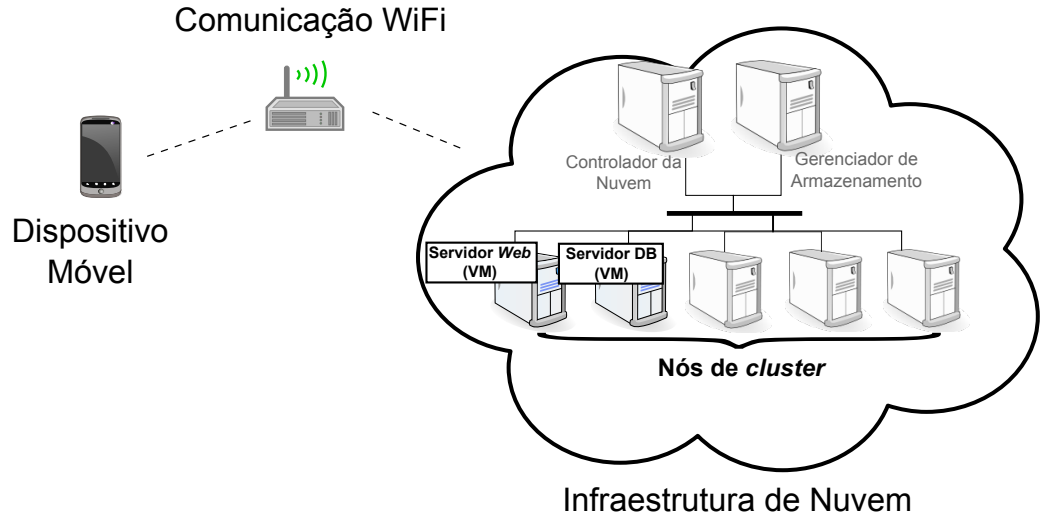


Figura 3.2: Arquitetura do sistema

3.2 Plataforma de Injeção de Falhas

A fim de validar o modelo proposto, criamos uma plataforma de testes para avaliação de disponibilidade em ambientes de *mobile cloud*. Ilustramos a arquitetura da plataforma na Figura 3.3. O sistema que estará sujeito à injeção de falhas é um protótipo de aplicação *Web* móvel, que consiste em um catálogo de livros *online*, e um *webservice* para consulta

de um livro através de seu ISBN. Ele é composto por um cliente móvel que executa em *smartphones Android* (Android 2013) e um *webservice* implantado em um servidor de aplicação *Glassfish* (Glassfish 2013). Este *webservice* faz consultas a uma base de dados em um servidor *PostgreSQL* de banco de dados (PostgreSQL 2013).

No dispositivo móvel, temos os seguintes componentes da plataforma de injeção de falhas:

- ***Workload Generator*** - Esta é a aplicação cliente do serviço em nuvem, que emula a utilização contínua do serviço por um usuário. Esta aplicação faz requisições sucessivas ao *webservice*, com um intervalo especificado (cinco segundos é o valor padrão adotado) entre as requisições. Cada resposta recebida é passada para o *Monitor Agent*;

- ***Monitor Agent*** - Este módulo é o cliente do serviço de monitoramento de disponibilidade, que atua em conjunto com o *Workload Generator*. Ele verifica cada mensagem recebida e analisa a integridade da mensagem. Se a mensagem de resposta contém algum erro, ou se não foi possível realizar a requisição, um estado de indisponibilidade é registrado em um arquivo de *log*. Caso contrário, ele escreve no *log* que o sistema estava disponível naquele momento. O *Monitor Agent* é consultado periodicamente pelo *Availability Monitor*, e retorna como resposta os conteúdos do *log*.

As atividades de injeção de falhas e monitoramento são executadas fora do dispositivo móvel, em um computador conectado ao dispositivo através de um cabo USB e rede WLAN. Este computador executa os seguintes módulos:

- ***Android Debug Bridge*** - O *Android Debug Bridge* (ADB) é parte do kit de desenvolvimento para a plataforma *Android*, e permite manipular um dispositivo *Android* conectado via USB, através de comandos específicos do sistema operacional *Android*, e alguns comandos *GNU/Linux*;

- ***Android Fault Injector*** - Este módulo foi construído para emitir comandos através do ADB, de forma controlada. Tais comandos caracterizam os eventos de falha e reparo dos componentes do dispositivo móvel. Cada componente (*hardware*, SO, aplicação, interface *wireless*) é manipulado por uma *thread* dedicada. O

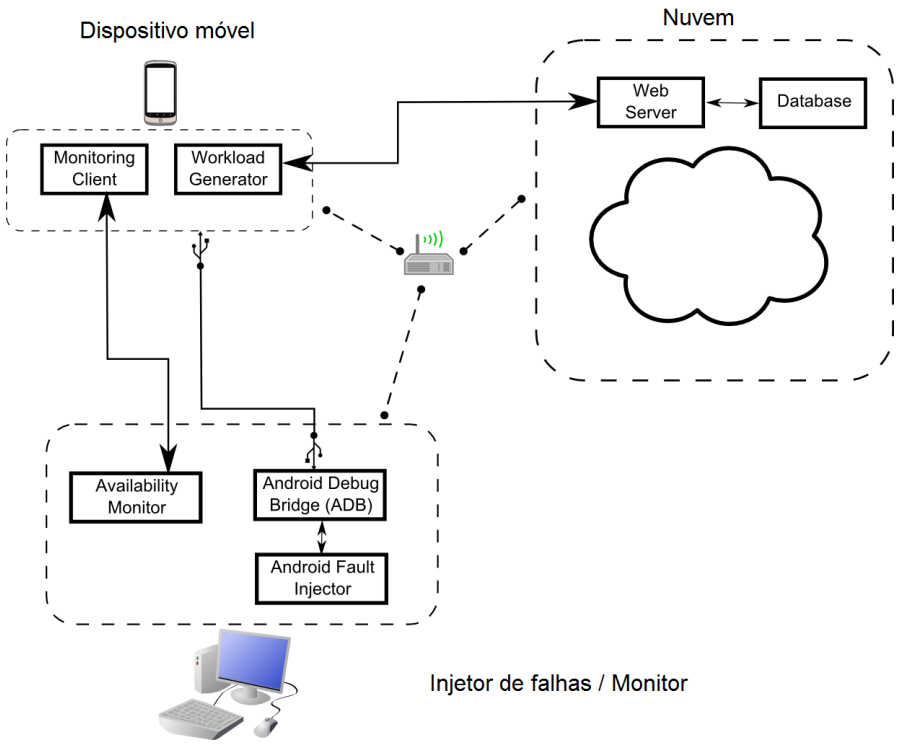


Figura 3.3: Arquitetura da plataforma de testes

comportamento das *threads* é descrito pelo pseudocódigo abaixo:

while *não interrompido* **do**
    suspenda a thread com tempo aleatório exponencial de média MTTF;
    falhe o componente;
    suspenda a thread com tempo aleatório exponencial de média MTTR;
    repare componente;
end

Algorithm 1: Pseudocódigo da *thread* de injeção de falhas

- ***Availability Monitor*** - Este módulo realiza o monitoramento da disponibilidade do sistema, do ponto de vista do cliente móvel. Ele envia requisições regularmente ao módulo *Monitor Agent* de acordo com um intervalo especificado (o padrão é de cinco segundos), obtém o *log* gerado por ele, e adiciona as entradas ao seu próprio *log*. Quando ele não é capaz de contatar o cliente móvel, seja por uma falha no dispositivo, aplicação, ou conexão *wireless*, um evento de indisponibilidade é registrado.

No lado da nuvem, temos dois nós físicos atuando como *host* para duas máquinas virtuais. Nestas máquinas virtuais são implantados o servidor *web Glassfish* (Glassfish

2013), e o banco de dados *PostgreSQL* (PostgreSQL 2013). O *hypervisor* sobre o qual as máquinas virtuais executam é o KVM [1]. A fim de injetar falhas nos componentes da nuvem, nós usamos o *script Perl* listado no Apêndice A. Usamos uma cópia modificada do *script* para cada componente que sofrerá injeção de falhas e reparos. As variáveis $\$command\_fail$ e $\$command\_repair$ designam os comandos específicos que provocam a falha e o reparo do componente. As variáveis $\$fail\_time$ e $\$repair\_time$ representam os tempos médios entre falhas e entre reparos, respectivamente, assumindo que esses tempos são exponencialmente distribuídos.

3.3 API de Simulação

Para complementar o processo de validação dos modelos realizado através dos experimentos de injeção de falhas, nós também provemos *scripts* de simulação baseados no NS-2 (Meenaghan & Delaney 2004). O Network Simulator (versão 2), mais conhecido como NS-2, é um simulador de redes baseado em eventos discretos que foi construído a partir do simulador de rede REAL, em 1989. Desde então, vários pesquisadores e instituições como a *Defense Advanced Research Projects Agency* (DARPA) e *National Science Foundation* (NFS) contribuíram para seu desenvolvimento. Hoje ele é o simulador de redes de código aberto mais usado na área acadêmica e de pesquisa (Ingalls 2008).

O NS-2 provê mecanismos para simulações de redes sem fio e cabeadas, aplicações, modelos de tráfegos, algoritmos de roteamento, protocolos *multicast*, etc. Com ele é possível modelar e avaliar o desempenho de redes de computadores através de simulações, validar protocolos e também pode ser utilizado para fins educativos. Ele também provê um *framework* básico para a criação de modelos de simulação baseados em eventos discretos. Este *framework* é composto por: i) um *clock* de simulação; ii) um agendador de eventos; iii) geradores de números aleatórios. Desta forma, o NS-2 também pode ser utilizado como uma ferramenta de simulação de propósito geral, embora o uso comum dele seja para a simulação de redes de computadores.

Em nossa extensão do NS-2 para simulação de análise de disponibilidade nós não usamos as capacidades de simulação de rede. Utilizamos apenas o *framework* básico de simulação: *clock* de simulação, agendador de eventos e geradores de números aleatórios. A Figura 3.4 mostra o diagrama de classe da API de simulação implementada na linguagem OTcl (Wetherall 2009), que é a linguagem de *scripting* do NS-2.

Em nossa API de simulação nós temos uma classe abstrata chamada *Component*, que

[1] http://www.linux-kvm.org/page/Main_Page

descreve um componente do sistema que pode eventualmente falhar e ser reparado. Esta classe é especializada em duas subclasses: *SingleComponent* e *CompoundComponent*. Uma instância da classe *SingleComponent* representa um componente cujos eventos de falha e reparo são escalonados no NS-2 através da geração de números aleatórios. A partir desta classe, nós derivamos subclasses que indicam distribuições de probabilidades em particular que descrevem seus MTTF e MTTR. Foram criadas duas subclasses, *WeibullComponent* e *ExponentialComponent*, que representam as distribuições *Weibull* e exponencial, respectivamente. Outras subclasses para distribuições diferentes poderiam ser incluídas na API, conforme a necessidade.

A outra subclasse que deriva de *Component* é chamada *CompoundComponent*. Uma instância desta subclasse representa um componente que é formado pela agregação de outros componentes. Estes subcomponentes podem ser instâncias de *SingleComponent* ou de *CompoundComponent*, permitindo vários níveis de aninhamento. O estado operacional de um componente composto é dependente dos eventos de falha e reparo de seus subcomponentes. Por exemplo, uma instância de *SeriesComponent* falhará caso algum dos seus subcomponentes falhe. Por outro lado, uma instância de *ParallelComponent* permanecerá operacional até que todos seus subcomponentes entrem em um estado de falha.

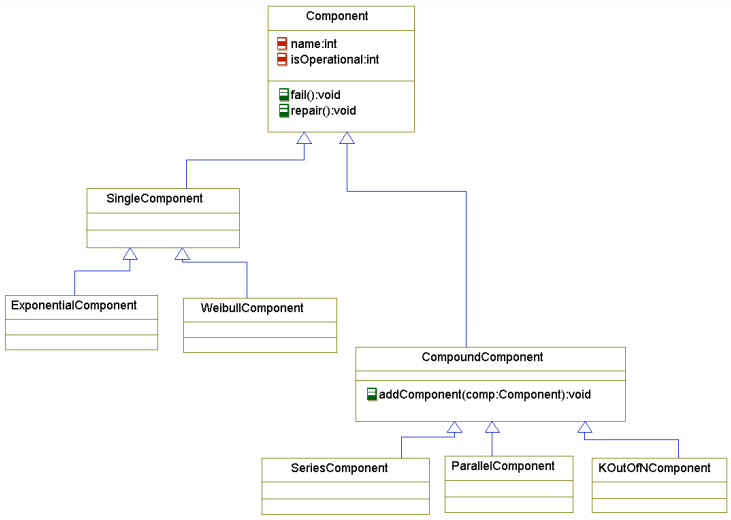


Figura 3.4: Diagrama de classe para a API de simulação

Ilustraremos o uso da API de simulação através de um exemplo adaptado de (Dantas et al. 2012*a*). Este cenário consiste em uma infraestrutura de nuvem privada composta por três nós de cluster, responsáveis por instanciar as máquinas virtuais da nuvem, um controlador de *cluster* (CLC) e um controlador da nuvem (CC) (uma discussão mais

detalhada sobre estes componentes é encontrada na seção 4.5). O modelo em RBD para este cenário está ilustrado na Figura 3.5. O *script* de simulação equivalente ao modelo é mostrado na Listagem 3.1. Nas primeiras linhas, importamos a API de componentes (*components.tcl*), e instanciamos o objeto de simulação do NS-2. Em seguida, criamos os componentes para cada bloco do RBD, os três nós de cluster, o CC e o CLC (nas linhas de 4 a 8). O construtor recebe como parâmetros o nome do bloco, o objeto de simulação, o MTTF e o MTTR, respectivamente. Em seguida, criamos cada agrupamento de blocos. Inicialmente criamos um componente paralelo com os três nós de cluster, e adicionamos os nós ao componente (linhas 10 e 11). Por último, criamos um componente em série formado por este componente paralelo, o CC e o CLC (linhas 13 e 14). Nas linhas de 16 a 20, criamos uma rotina de finalização para a simulação, que imprime a disponibilidade computada pelo modelo. A linha 22 cria o agendamento desse evento de finalização no objeto de simulação e, finalmente, na linha 24, iniciamos o processo de simulação.

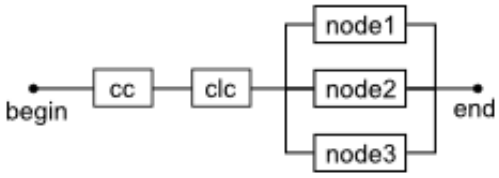


Figura 3.5: Exemplo de modelo para nuvem privada em RBD

Listagem 3.1: *Script* de simulação

```
1  source components.tcl
2  set ns [new Simulator]
3
4  set node01 [new ExponentialComponent Node01 $ns 800 2]
5  set node02 [new ExponentialComponent Node02 $ns 800 2]
6  set node03 [new ExponentialComponent Node02 $ns 800 2]
7  set clc [new ExponentialComponent ClusterController $ns 1500 1]
8  set cc [new ExponentialComponent CloudController $ns 1500 1]
9
10 set nodes [new ParallelComponent Nodes $ns]
11 $nodes addComponents $node1 $node2 $node3
12
13 set cloud [new SeriesComponent Cloud $ns]
```

```
14 $cloud addComponents $clc $cc $nodes
15
16 proc finish {} {
17         global ns cloud
18         puts [$cloud availability]
19         $ns halt
20 }
21
22 $ns at 10000000 "finish"
23
24 $ns run
```

A API de simulação e o *script* utilizado para o modelo de simulação da arquitetura básica estão listados no Apêndice B.

4

Arquiteturas e Modelos

Neste capítulo, apresentamos as arquiteturas avaliadas e os modelos criados para a avaliação da disponibilidade destas arquiteturas. Inicialmente descrevemos o modelo da arquitetura básica, que é introduzida na Seção 3.1, e detalhamos seus componentes. Em seguida, mostramos extensões da arquitetura básica que tem como objetivo o aumento da disponibilidade estacionária. As alternativas arquiteturais consideradas são: arquitetura *store-and-forward*, arquitetura de múltiplas interfaces de rede e arquitetura *cloudlet*. Na Seção 4.5, apresentamos um mecanismo de redundância a ser incorporado na nuvem, que visa melhorar a disponibilidade das arquiteturas consideradas.

Na Seção 4.6, expomos um modelo de disponibilidade de uma *mobile cloud* que leva em consideração os efeitos das interfaces *wireless* sobre a taxa de descarga da bateria. Também propomos um modelo capaz de calcular o tempo médio de descarga da bateria, de acordo com a proporção de tempo que o cliente mantém cada interface *wireless* ativada.

4.1 Arquitetura base

Na Figura 4.1, exibimos o modelo para a arquitetura básica descrita na Seção 3.1. No topo da Figura 4.1, nós representamos o modelo alto nível RBD para o sistema. O modo de falha deste modelo é caracterizado pela composição em série dos seguintes subsistemas: dispositivo móvel, conexão wireless e infraestrutura em nuvem. Todos estes subsistemas deverão funcionar para que o sistema esteja disponível. O sombreamento de blocos no modelo indica que aquele componente é representado por outro submodelo de forma hierárquica. Estes submodelos são descritos na figura abaixo do modelo de alto nível.

O modo de falha do dispositivo móvel é representado pela falha de algum dos seguintes componentes: *hardware* do dispositivo móvel, sistema operacional, bateria e aplicação móvel. A falha da conexão *wireless* é constituída pela falha do ponto de acesso *wireless*,

ou pelo bloqueio do sinal wireless devido à obstáculos físicos (Chen et al. 2003).

No nosso modelo, o modo de falha da bateria é causado por sua descarga. Um modelo em CTMC, descrito na Figura 4.2, representa o processo de descarga da bateria em passos de 10%, que ocorrem sob taxa dw. O estado "0" está destacado para indicar que este é um estado no qual a bateria está indisponível, enquanto que todos os outros são estados onde a bateria está disponível. O tempo médio de descarga de 10% da bateria, ou seja, $1/dw$, é assumido ser 0.9 horas. Este modelo assume a existência de uma bateria reserva, que é usada para substituir a bateria em uso quanto esta é completamente descarregada. A taxa de substituição é rb, o que significa que o tempo médio para substituição após uma descarga é de $1/rb$. O tempo médio de substituição é considerado de 5 minutos devido o atraso na substituição manual, tempo de inicialização do sistema operacional e da aplicação.

O serviço em nuvem é representado por dois blocos em série que correspondem aos nós que fornecem o serviço para o cliente móvel: o servidor *Web* e o servidor de banco de dados. Cada nó é representado por um modelo RBD com cinco componentes em série: *hardware* do nó, sistema operacional do nó, o *software* de virtualização KVM, sistema operacional virtualizado e a aplicação que executa sobre o sistema operacional virtualizado.

A disponibilidade desta arquitetura básica de *mobile cloud* pode ser aumentada, uma vez que esta arquitetura não contempla mecanismos de redundância. A falha de qualquer componente do modelo da Figura 4.1 é capaz de levar o sistema à indisponibilidade. Nas próximas seções, investigamos algumas variações da arquitetura básica com o objetivo de aumentar a disponibilidade, através de melhorias na conectividade *wireless* e na nuvem.

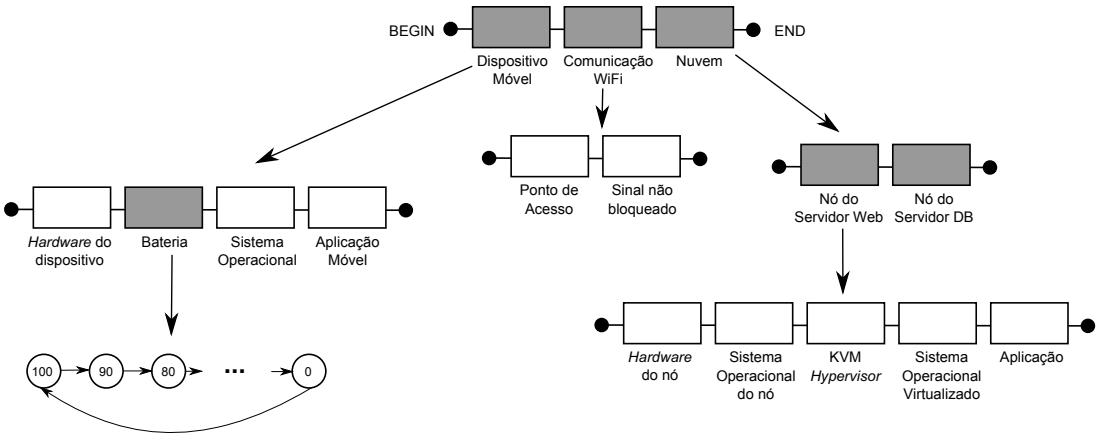


Figura 4.1: Modelo RBD hierárquico para a arquitetura básica

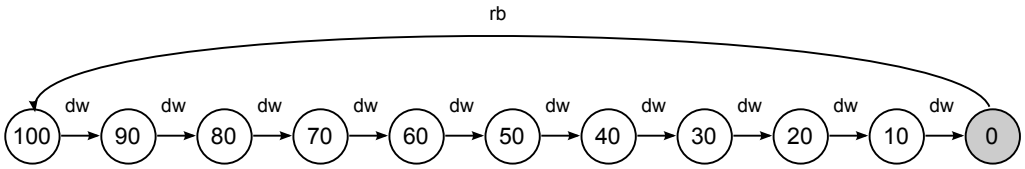


Figura 4.2: Diagrama de estados da CTMC para a bateria do dispositivo móvel

4.2 Arquitetura *Store and Forward*

Conectividade *wireless* será sempre um fator crítico para a dependabilidade de *mobile clouds*. Ao mesmo tempo em que a *cloud computing* ajuda a aumentar o poder da computação móvel, ela impõe uma demanda de uma conexão *wireless* sempre ativada. Um cliente móvel que dependa de serviços em nuvem se torna indisponível se esta conexão for interrompida.

Em alguns casos, podemos lidar com este problema removendo a exigência de uma conexão sempre ativada. Em nossa aplicação do *testbed* de injeção de falhas, a aplicação de uma livraria móvel, considere um operador do sistema que apenas registra novos livros do estoque. Ele poderá usar o sistema mesmo se a conexão com a nuvem esteja indisponível, gravando os novos registros na memória do dispositivo. Quando a conexão se torna novamente disponível, o cliente móvel realiza a operação de sincronização com a nuvem e atualiza os novos registros. Este estilo arquitetural é denominado *store and forward* (H. Schneider & Schell 2004). Esta estratégia tem como vantagem oferecer uma maior disponibilidade que outras que dependem de uma conexão com a nuvem, porém, algumas desvantagens podem impedir sua implementação. Por exemplo, se o operador precisa buscar por informações adicionais sobre um livro escaneando seu código de barras, será necessário acessar a nuvem. A arquitetura *store and forward* não ajudaria muito neste caso. Uma possível solução para este problema seria manter uma réplica da base de dados da nuvem no dispositivo móvel, e manter a réplica sincronizada sempre que a conexão estivesse disponível. Contudo, dois problemas surgem com esta solução. O primeiro problema é a impossibilidade de manter uma réplica de uma grande base de dados, devido a limitações de espaço de armazenamento em dispositivos móveis. O segundo problema são os problemas de integridade de dados e inconsistências que surgem naturalmente ao utilizar estas estratégias de replicação.

O modelo de disponibilidade para esta arquitetura é descrito na Figura 4.3. Este modelo é semelhante ao modelo para o dispositivo móvel da Figura 4.1, com a adição de

um componente: um dispositivo de armazenamento secundário onde ficam armazenados os dados sincronizados com a nuvem. O componente *Bateria* está sombreado para indicar o fato que ele é avaliado pelo modelo em CTMC da Figura 4.2. A fórmula de disponibilidade para este modelo é apresentada na Equação 4.1. Os valores de $A_{mob\_hardware}$, $A_{bateria}$, A_{SO}, A_{mob_app}, $A_{storage\_card}$ correspondem à disponibilidade dos componentes: *hardware* do dispositivo móvel, bateria, sistema operacional móvel, aplicação móvel e cartão de armazenamento, respectivamente.

$$A = A_{mob\_hardware} \times A_{bateria} \times A_{SO} \times A_{mob\_app} \times A_{storage\_card} \tag{4.1}$$

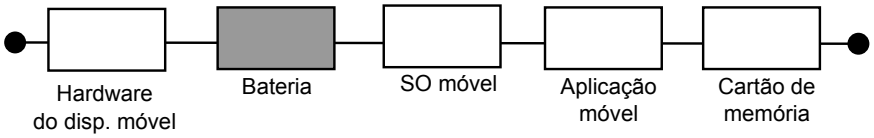


Figura 4.3: Modelo RBD para arquitetura *store and forward*

4.3 Múltiplas Interfaces de Rede

Se o uso da arquitetura *store and forward* não for aplicável pelos motivos expostos na seção anterior, a existência de uma conexão *wireless* sempre ativada se faz necessária. Desta forma, uma maneira de aumentar a disponibilidade é permitir a utilização de uma segunda interface de rede sem fio. Esta interface redundante pode evitar a desconexão através dos seguintes modos: 1) habilitando a conexão a diferentes redes com a mesma tecnologia (por exemplo: duas redes WiFi) e, desta forma, protegendo o sistema de um único ponto de falha na rede sem fio; 2) habilitando conexão sem fio através de diferentes tecnologias (por exemplo: uma conexão WiFi e outra 3G), permitindo uma maior mobilidade e reduzindo a probabilidade de desconexão devido a sinal bloqueado ou fraco.

A Figura 4.4 mostra o modelo RBD para a arquitetura de múltiplas interfaces de rede, considerando uma interface WiFi e outra 3G. As interfaces sem fio redundantes são representadas por blocos em paralelo no RBD. Os blocos restantes, *Dispositivo Móvel*, *Nuvem* e seus respectivos componentes são representados em modelos de níveis inferiores da mesma forma que na arquitetura básica, exceto pela bateria do dispositivo móvel, que é representada pela CTMC descrita na Figura 4.5. Este modelo é similar ao da CTMC apresentado na Figura 4.2, porém, introduz diferenças entre o processo de descarga ao

usar a interface 3G ou WiFi. O dispositivo inicia o processo de descarga enquanto usa a conexão WiFi com probabilidade pw e, com probabilidade $p3g = 1 - pw$, inicia o processo de descarga quando a conexão 3G está em uso. A taxa de descarga para 10% da capacidade total é de dw enquanto usa a conexão WiFi e $d3G$ enquanto usa a interface 3G. Note que a transição entre WiFi e 3G não é representada no modelo, por motivo de simplicidade. Assumimos que esta simplificação não produz impacto significante nos resultados do modelo completo.

A fórmula da disponibilidade para o modelo da Figura 4.4 é representada na Equação 4.2. Nesta equação, A_{device} representa a disponibilidade do dispositivo móvel e A_{cloud} a disponibilidade da nuvem. A_{wifi} e A_{3G} correspondem às disponibilidades das interfaces WiFi e 3G, respectivamente.

$$A = A_{device} \times (1 - (1 - A_{wifi}) \times (1 - A_{3g})) \times A_{cloud} \tag{4.2}$$

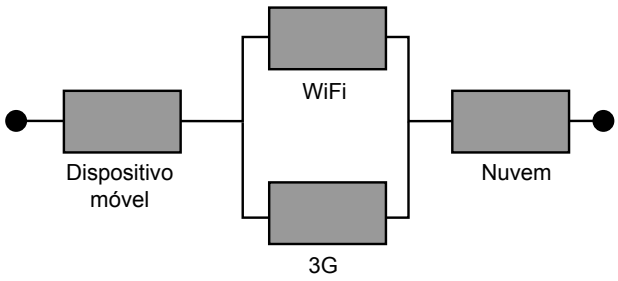


Figura 4.4: RBD para múltiplas interfaces de rede

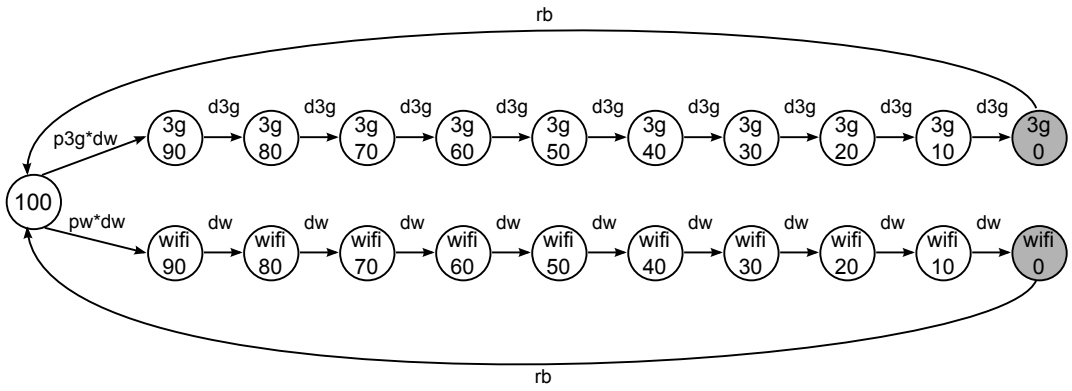


Figura 4.5: Modelo de descarga da bateria com WiFi e 3G

O uso de múltiplas interfaces de rede por um cliente de um serviço em nuvem trás consigo algumas questões a serem consideradas. A conectividade *wireless* é um dos

fatores que mais provocam impacto no consumo de energia por um dispositivo móvel (Kalic et al. 2012). Dado que diferentes interfaces *wireless* possuem diferentes larguras de banda, também será diferente o consumo energético causado pelo uso de tais interfaces. Uma interface que possui menor largura de banda demandará mais tempo para transmitir uma certa quantidade de dados, causando um maior consumo de energia. Desta forma, sempre que tivermos diferentes interfaces *wireless* disponíveis, devemos priorizar a que possui maior largura de banda, pois, além de aumentar o tempo de resposta da aplicação, ela irá causar um menor impacto no tempo de vida da bateria (Friedman et al. 2013).

O maior nível de consumo energético causado por uma interface *wireless* é atingido quando a interface está sendo usada para transmitir ou receber dados. Porém, o simples fato de deixar a interface ativada também provoca um aumento no consumo energético. Portanto, é recomendado que quando uma interface estiver habilitada para a transmissão de dados da aplicação, as outras sejam desligadas. Contudo, uma desvantagem surge ao utilizar esta estratégia. Quando uma conexão ativa é interrompida por bloqueio de sinal, por exemplo, a outra interface não está imediatamente pronta para assumir a transferência de dados. Há uma certa indisponibilidade durante o tempo necessário para ativar a outra interface, que vai desde a ativação do *hardware* da interface de rede, até o estabelecimento da conexão com o provedor de rede sem fio.

Essas características discutidas não são bem representadas por modelos combinatórios como RBDs. Assim, se quisermos investigar essas questões de forma mais profunda, será necessário o uso de um formalismo com maior poder de expressividade, como as cadeias de Markov ou redes de Petri estocásticas. Na Seção 4.5, nós apresentamos modelos em redes de Petri estocásticas que nos permitem avaliar o impacto do uso de múltiplas conexões sem fio sobre a disponibilidade e o consumo energético da aplicação móvel.

4.4 Cloudlet

Uma *cloudlet* é uma rede privada confiável, localizada em uma área com alta concentração de usuários, tais como: *shopping centers*, aeroportos, etc (Satyanarayanan et al. 2009). O principal objetivo de uma *cloudlet* é prover recursos computacionais para clientes móveis com um tempo de resposta mais baixo, uma vez que tais recursos estarão próximos dos usuários, na mesma WLAN. Desta forma, clientes podem obter os benefícios da computação em nuvem, sem sofrer dos problemas que surgem ao acessar uma nuvem remota via rede WAN, como atraso e *jitter*. Do ponto de vista da dependabilidade, a *cloudlet* também pode prover redundância, habilitando o serviço para o cliente até mesmo

quando a nuvem principal estiver indisponível.

A Figura 4.6 mostra um esquema de *cloudlet* e nuvem pública trabalhando juntas. Se o cliente permanece na área de cobertura da WLAN, onde a *cloudlet* está localizada, ele irá acessar os serviços da *cloudlet*. Isto o permitirá obter serviços com um menor tempo de resposta que o obtido através do serviço provido pela nuvem remota. Quando o cliente se move para fora do alcance da *cloudlet*, ele pode obter serviço da nuvem pública através de rede 3G, embora com maiores atrasos.

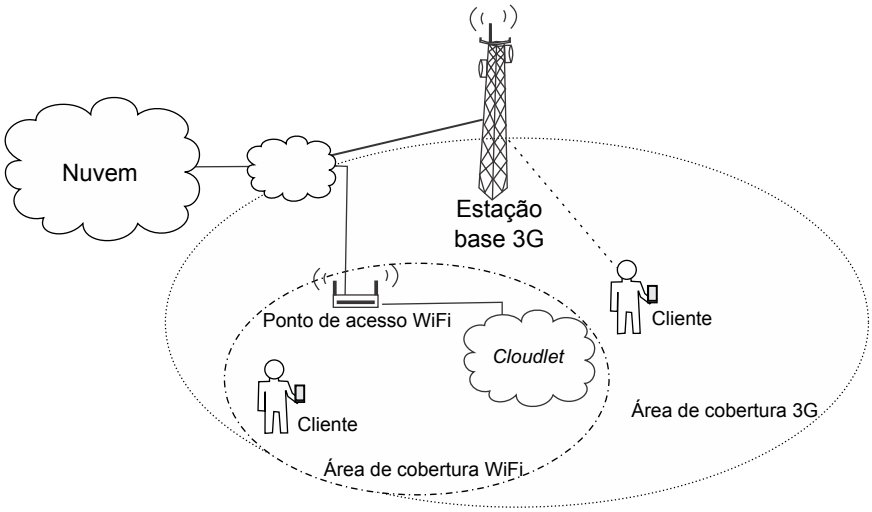


Figura 4.6: Arquitetura *cloudlet*

Se a *cloudlet* está indisponível, o cliente pode acessar a nuvem pública via rede WLAN ou conexão 3G. Por outro lado, se a nuvem pública está indisponível, ele pode receber serviço apenas se estiver próximo à *cloudlet*. A Figura 4.7 mostra o modelo de disponibilidade para esta solução. Observe que o bloco *Nuvem* ocorre em dois locais. Não se trata de duas *clouds* diferentes, mas de duas ocorrências do mesmo componente. Para resolver este modelo, devemos fatorar o componente repetido (Maciel et al. 2012), gerando dois submodelos. O primeiro submodelo (Figura 4.8a) representa o caso quando o componente *Nuvem* está indisponível. O segundo submodelo (Figura 4.8b) representa a situação inversa. A equação da disponibilidade do primeiro submodelo é dada pela Equação 4.3 e a+ do segundo é dada pela Equação 4.4.

$$A_1 = A_{\text{device}} \times (1 - (1 - A_{3g}) \times (1 - A_{\text{wifi}})) \tag{4.3}$$

$$A_2 = A_{\text{device}} \times A_{\text{wifi}} \times A_{\text{cloudlet}} \tag{4.4}$$

Desta forma, podemos obter a disponibilidade para a arquitetura de *cloudlet* através da seguinte fórmula mostrada na Equação 4.5.

$$A = A_{\text{cloud}} \times A_1 + (1 - A_{\text{cloud}}) \times A_2$$

$$A = A_{\text{device}} \times (A_{\text{cloudlet}} \times A_{\text{wifi}} + A_{\text{cloud}} \times (A_{3g} - (-1 + A_{\text{cloudlet}} + A_{3g}) \times A_{\text{wifi}})) \quad (4.5)$$

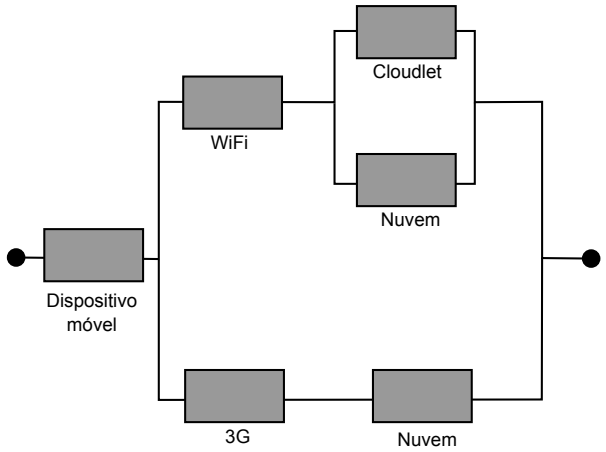
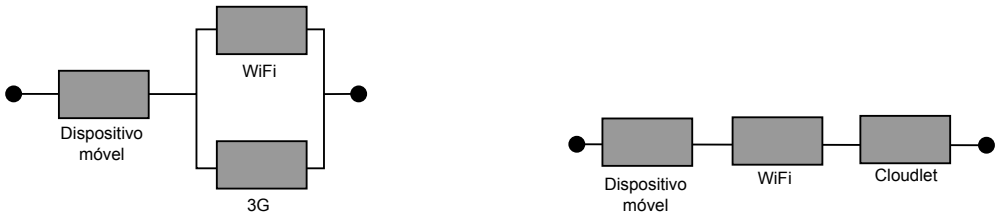


Figura 4.7: Modelo RBD para arquitetura *cloudlet*



(a) Componente *Nuvem* disponível

(b) Componente *Nuvem* indisponível

Figura 4.8: Fatoração do modelo em RBD da arquitetura *cloudlet*

4.5 Redundância nos Nós da Nuvem

Observando a nuvem no modelo em RBD da arquitetura básica, percebemos que ela é caracterizada por uma série de componentes onde a falha de qualquer um destes provoca a indisponibilidade do lado da nuvem. Em ambientes de nuvem pública ou privada, a infraestrutura de nuvem é formada por vários nós de *cluster* e alguns nós que executam

módulos de *software* responsáveis pelo gerenciamento da nuvem. Caso um nó de *cluster* sofra uma falha, a máquina virtual que estava executando neste nó pode ser implantada em outro nó de *cluster* disponível, seja de forma manual por um administrador da nuvem, ou de forma automatizada caso exista uma implementação desta funcionalidade.

Alguns *frameworks* como *Eucalyptus* (Eucalyptus 2013), *OpenStack* (OpenStack 2013) e *OpenNebula* (OpenNebula 2013) podem ser usados para implementar um ambiente de infraestrutura como serviço (IaaS) (Peng et al. 2009). Os *frameworks* que implementam um estilo de modelo IaaS privado/híbrido tem componentes similares com nomenclaturas ligeiramente diferentes. Nosso modelo considera as principais características de vários *frameworks* de nuvem, tal que os componentes recebem nomes genéricos para enfatizar que o modelo não está ligado a nenhum *framework* específico.

Podemos visualizar a infraestrutura de nuvem privada com todos os componentes no RBD da Figura 4.9. Este modelo é dividido em três submodelos: *Infrastructure Manager* (IM), *Storage Manager* (SM) e o *Cluster*. O *Infrastructure Manager* corresponde ao nó físico onde executam os *softwares* de gerenciamento da infraestrutura de nuvem. Suas principais atribuições são monitorar e alocar recursos de um ou mais *clusters* e provisionar estes recursos para os clientes da nuvem. O *Storage Manager* provê mecanismos de armazenamento para a nuvem, com a finalidade de armazenar imagens de máquinas virtuais, *snapshots* de instâncias de VMs, dados de aplicações da nuvem, etc. Um *Cluster* é formado por um conjunto de nós que oferecem seus recursos (processamento, memória, etc.) para os clientes da nuvem através de virtualização. Nesta seção consideraremos o caso mais simples, onde a infraestrutura possui apenas um *cluster* com n nós.

Em nossa arquitetura consideramos que cada VM é executada em um nó diferente do outro, portanto, por questão de consistência, iremos manter a mesma suposição neste modelo. De N nós de *cluster* da infraestrutura, consideramos que é necessário que pelo menos dois estejam disponíveis. Uma falha no IM ou SM não afeta diretamente a disponibilidade do ponto de vista do cliente, porém, afeta indiretamente, pois perdemos a capacidade de administrar a infraestrutura e implantar novas VMs.

O submodelo para o *Infrastructure Manager* é descrito na Figura 4.10. Ele possui quatro componentes: *HW*, *OS*, *CLC* and *CC*. *HW* representa o *hardware* físico; *OS* é o sistema operacional usado pela máquina; *CLC* se refere ao *Cloud Controller*, isto é, o componente principal responsável por gerenciar toda a infraestrutura de nuvem; *CC* representa o *Cluster Controller*, que atua como um *front-end* de *cluster* e gerencia um conjunto de nós.

O *Storage Manager*, ilustrado na Figura 4.11, é dividido em cinco componentes:

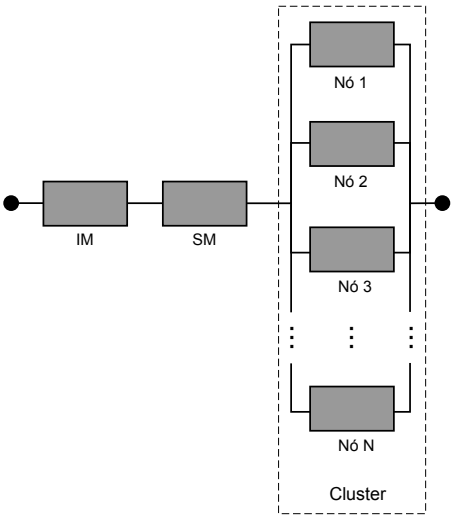


Figura 4.9: Modelo em RBD para a infraestrutura de nuvem privada

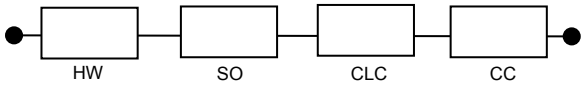


Figura 4.10: Modelo em RBD para o subsistema *Infrastructure Manager*

HW, *OS*, *FSC*, *BSC* e *NAS*. *HW* e *OS* são o *hardware* e sistema operacional do nó SM, respectivamente; *FSC* é o *File-based Storage Controller*, que representa um serviço de armazenamento baseado em arquivos; *BSC* significa *Block-based Storage Controller*, que provê armazenamento persistente de bloco para o armazenamento de instâncias de máquinas virtuais; por último, o *NAS* se refere ao *Network-Attached Storage*, que é um dispositivo que provê armazenamento de dados da rede para os controladores de armazenamento da nuvem.

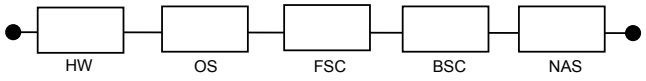


Figura 4.11: Modelo RBD para o subsistema *Storage Manager*

O modelo para um nó de *cluster* (Figura 4.12) é semelhante ao do modelo visto anteriormente, com a adição de um novo componente de *software* chamado *Node Controller (NC)*. Este módulo é responsável por gerenciar o nó e aceitar comandos de gerenciamento do CC e do CLC, como por exemplo, o de instanciar uma VM. Neste nó executa também a

camada de virtualização, que em nossa arquitetura representamos pelo componente KVM. Além disso, representamos neste modelo o sistema operacional virtualizado (OS_VM) e a aplicação que executa nesta máquina virtual (APP_VM).

A Equação 4.6 mostra a fórmula da disponibilidade para o modelo da Figura 4.9. Os valores de A_{IM}, A_{SM} e A_{node_i} são obtidos através das equações 4.7, 4.8 e 4.9, respectivamente.

$$A = A_{IM} \times A_{SM} \times [1 - \prod_{i=1}^{n}(1 - A_{node_i})] \tag{4.6}$$

$$A_{IM} = A_{HW} \times A_{OS} \times A_{CC} \times A_{CLC} \tag{4.7}$$

$$A_{SM} = A_{HW} \times A_{OS} \times A_{FSC} \times A_{BSC} \times A_{NAS} \tag{4.8}$$

$$A_{node_i} = A_{HW} \times A_{OS} \times A_{KVM} \times A_{NC} \times A_{OS\_VM} \times A_{APP\_VM} \tag{4.9}$$

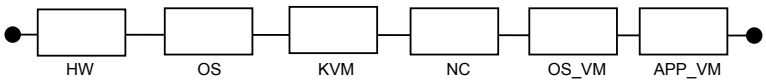


Figura 4.12: Modelo RBD para cada nó do subsistema *Cluster*

O modelo da Figura 4.9 não reflete a disponibilidade se analisarmos do ponto de vista do cliente móvel do serviço em nuvem. Para esta finalidade, precisamos de um formalismo com maior poder, como por exemplo, cadeias de Markov ou redes de Petri estocásticas. Optando pela segunda opção, construímos o modelo que está representado na Figura 4.13. Neste modelo, os lugares principais são *web_server* e *db_server*, que representam os nós de *cluster* onde estão implantadas as VMs do servidor web e do servidor de banco de dados, respectivamente. A presença de um *token* nestes lugares representa o funcionamento correto destes nós e das VMs implantadas. O lugar *cluster_machines_up* representa os nós de *cluster* que estão funcionando corretamente. O número de marcas neste lugar é um parâmetro do modelo que varia de 0 até k. Caso este parâmetro seja zero, a infraestrutura de nuvem privada terá dois nós de cluster, um para o *web server*, outro para o banco de dados. À medida que aumentamos este parâmetro, teremos mais máquinas a disposição para implantar as VMs do serviço, caso ocorram falhas. A transição $cluster\_machine\_fail$ representa a falha de uma máquina

do *cluster* (que não está sendo usada pelo *web server* ou banco de dados). Esta transição retira *tokens* de *cluster_machines_up* e coloca-os em *cluster_machines_failed*, que representa as máquinas do *cluster* que falharam e estão aguardando manutenção. A transição *cluster_machine_repair* representa a operação de reparo, que coloca os *tokens* de volta para o *cluster_machines_up*. As transições *db_server_fail* e *web_server_fail* representam a falha das máquinas específicas onde estão alocadas as VMs descritas anteriormente.

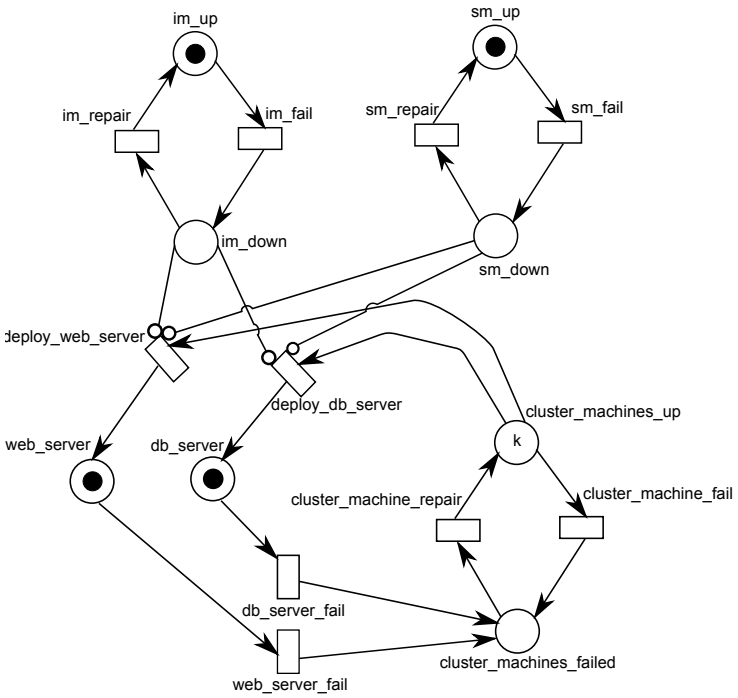


Figura 4.13: Modelo em rede de Petri para infraestrutura de nuvem privada

Caso ocorra alguma falha nas VMs de serviço *web*, ou banco de dados, a infraestrutura de nuvem privada seleciona um nó de *cluster* que esteja disponível, e faz a implantação da VM neste novo nó. No modelo, isto é representado pelas transições *deploy_web_server* e *deploy_db_server*, que retiram um *token* de *cluster_machines_up* e coloca-o em *web_server* ou *db_server*, respectivamente. Estas operações só podem ser realizadas se os nós IM e SM estiverem funcionando. O funcionamento destes nós é representado no modelo pelos lugares *im_up* e *sm_up*. Os eventos de falhas destes correspondem às transições: *im_fail* e *sm_fail*, enquanto que as operações de reparo correspondem às transições *im_repair* e *sm_repair*, respectivamente. A falha destes componentes é indicada pela presença de um *token* nos lugares *im_down* e *sm_down*. A presença de *token* nestes lugares impedem o disparo das transições de implantação de VMs, graças aos arcos inibidores ligados nestas transições.

A fórmula de disponibilidade da nuvem do ponto de vista do cliente móvel está descrita na Equação 4.10:

$$\mathrm{A} = P\{\#web\_server = 1 \text{ AND } \#db\_server = 1\} \tag{4.10}$$

É importante ressaltar que este mecanismo de redundância provoca mudanças significativas no serviço em nuvem da arquitetura básica, introduzindo uma série de novos componentes. Desta forma, não podemos considerar validado o modelo apresentado nessa seção. Como trabalho posterior, criaremos o mecanismo de redundância que atuará em conjunto com um *software* de nuvem privada responsável por esse mecanismo de redundância e avaliar sua efetividade no aumento da disponibilidade através de experimentos de injeção de falhas.

4.6 Modelos de Disponibilidade e Consumo Energético em Redes de Petri

Para investigar os efeitos do uso da conectividade sem fio sobre o consumo energético e a disponibilidade de uma aplicação, selecionamos a arquitetura *cloudlet* descrita na Seção 4.4 e criamos um modelo que reflete o efeito combinado da conectividade e consumo energético sobre a disponibilidade do sistema no cliente móvel. Uma vez que um modelo em RBD não é capaz de representar tais características, o uso de modelo baseados em estados é mais adequado nesta situação. Considerando o grande número de estados necessários para representar um cenário complexo como este, optamos pelo formalismo das Redes de Petri Estocásticas Generalizadas, em detrimento de cadeias de Markov. O modelo criado permite representar as seguintes características do sistema: conectividade *wireless* no dispositivo móvel; descarga da bateria em função do tipo de interface *wireless* utilizada; interação com nuvem remota e *cloudlet*. O modo operacional do sistema obedece às seguintes regras:

1. O sistema estará operacional se o dispositivo, sistema operacional e aplicativo móvel estiverem funcionais; se o cliente estiver conectado à nuvem remota ou à *cloudlet*; e se a bateria do aparelho possuir carga suficiente;

2. O cliente estará apto a se conectar à nuvem remota se possuir uma conexão WiFi ou 3G ativa e a nuvem remota estiver disponível;

3. O cliente estará apto a se conectar à *cloudlet* se possuir uma conexão WiFi ativa e se a *cloudlet* estiver disponível;

A Figura 4.14 apresenta o modelo criado. A parte superior do modelo da Figura 4.14 corresponde ao funcionamento das interfaces *wireless*: WiFi e 3G. Os lugares ap_up e ap_down definem o status do access point. O disparo da transição ap_fl provoca a falha do *access point*, removendo um token do lugar ap_up e movendo-o para ap_down. Após a falha do *access point*, é iniciado o reparo do mesmo, pela transição ap_rp. Os lugares $wifi\_nonblocking$ e $wifi\_blocking$ descrevem a situação do cliente em relação à presença de obstáculos que bloqueiam o sinal WiFi. Um *token* no lugar $wifi\_nonblocking$ indica que ele está numa região não bloqueante. Caso contrário, haverá um *token* em $wifi\_blocking$, indicando que o cliente está inapto a estabelecer uma conexão WiFi com o AP. Os lugares $wifi\_on$ e $wifi\_off$ informam o status do funcionamento do WiFi. A condição para a interface WiFi estar funcionando corretamente é que o AP esteja operacional e o cliente esteja em uma região não bloqueante. Caso algum *token* seja removido dos lugares ap_up ou $wifi\_nonblocking$, a transição imediata $disable\_wifi$ dispara e transfere o *token* de $WiFi\_up$ para $WiFi\_down$. Após as condições para a conexão WiFi ser estabelecida sejam novamente satisfeitas, será habilitada a transição temporizada $activate\_wifi$, que representa o tempo de ativação do WiFi. Uma vez que a conexão WiFi seja interrompida, o cliente tenta ativar a interface de rede 3G, através da transição $activate\_3G$. O comportamento do 3G no modelo é idêntico ao WiFi, sendo diferenciado pelos tempos médios de disparo das transições. Quando o cliente restabelece a conexão WiFi, a 3G é desativada, por questões de desempenho e consumo energético.

Na parte intermediária do modelo, é descrito o comportamento da bateria. O lugar $battery\_charge$ possui 100 *tokens*, representando a carga completa da bateria. A medida que a aplicação executa, consumindo energia da bateria, o modelo retira *tokens* desse lugar representando o processo de descarga da bateria. O tempo para descarga de 1% do total foi obtido através de experimentação.

O experimento consistiu no monitoramento da variação do percentual de carga da bateria a medida que o dispositivo executa o gerador de carga descrito na Seção 3.2. Cada mudança de status do percentual da bateria foi monitorado e registrado em um *log*. A taxa de descarga da bateria depende de vários fatores, um destes é a interface de rede utilizada. Uma mesma quantidade de dados pode demandar mais energia para ser transmitida, dependendo da interface utilizada. Em nosso experimento, foi medido o consumo energético causado pelo gerador de carga através do uso da interface WiFi. Para encontrar a taxa de descarga da interface 3G, utilizamos o modelo linear proposto

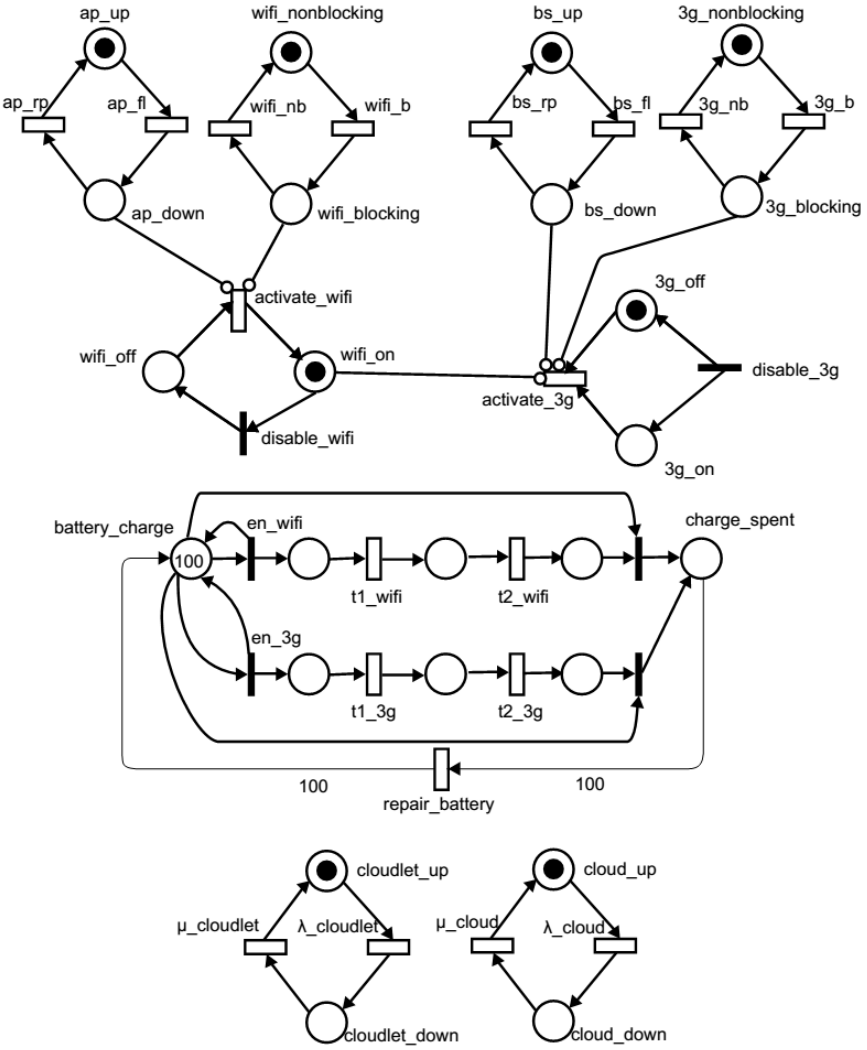


Figura 4.14: Rede de Petri estocástica para a disponibilidade de uma *mobile cloud*, com múltiplas interfaces de rede e *cloudlet*

por (Balasubramanian et al. 2009). Uma vez que os valores obtidos não podem ser representados adequadamente por uma distribuição exponencial, é necessário aproximá-los por um subconjunto de transições exponenciais. Empregamos a técnica descrita em (Watson & Desrochers 1991) para aproximação por distribuições *phase-type*, e obtemos uma distribuição hipo-exponencial de duas fases. Quando todos os *tokens* são retirados de *battery_charge* e chegam ao lugar *charge_spent*, eles são repostos aos lugares de origem através da transição *repair_battery*. Assumimos no nosso modelo que o usuário possui uma bateria completamente carregada para substituir a atual.

Por último, a parte inferior do modelo representa o status da nuvem remota e da *cloudlet*. Os lugares *cloudlet_up* e *cloud_up* indicam um funcionamento correto dos

serviços da *cloudlet* e da nuvem remota, respectivamente. Estes serviços possuem tempos de falha $\lambda\_cloudlet$ e λ_cloud, e tempos de reparo $\mu\_cloudlet$ e μ_cloud. Estes valores foram calculados na ferramenta SHARPE utilizando os modelos da Figura 4.1.

A disponibilidade do sistema do ponto de vista do cliente é calculado através da Equação 4.11:

$$A_{System} = A_{Android\_system} \times A_{Mobile\_cloud} \times A_{Battery}, \tag{4.11}$$

onde:

$$\begin{aligned} A_{Mobile\_cloud} = \\ & P\{(\#\text{wifi\_on} = 1 \text{ AND } \#\text{cloud\_up} = 1) \text{ OR} \\ & (\#\text{wifi\_on} = 1 \text{ AND } \#\text{cloudlet\_up} = 1) \text{ OR} \\ & (\#\text{3g\_on} = 1 \text{ AND } \#\text{cloud\_up} = 1)\}, \end{aligned} \tag{4.12}$$

e

$$A_{Battery} = P\{\#\text{battery\_charge} > 0\}\ . \tag{4.13}$$

Expressamos as Equações (4.12) e (4.13) de acordo com a notação usada pelas ferramentas TimeNet e Mercury. A_{Mobile_cloud} (Equação 4.12) é a probabilidade de haver uma conexão *wireless* (3G ou WiFi) com o serviço provido pela nuvem remota ou pela *cloudlet*. $A_{Battery}$ (Equação 4.13) corresponde a probabilidade estacionária de que o lugar *#battery_charge* tenha um número de *tokens* maior que zero, que representa à disponibilidade da bateria. $A_{Android}$ é a disponibilidade do sistema *Android*, conforme descrito no submodelo da Figura 4.1, exceto pelo componente da bateria, que já está representado de forma mais fidedigna na rede de Petri estocástica.

O modelo em rede de Petri estocástica da Figura 4.14 assume que o usuário sempre possui uma bateria reserva. Esta suposição é necessária para permitir uma alta disponibilidade, devido ao longo tempo de carregamento de uma bateria. Entretanto, a premissa de substituição da bateria poderá não ser válida em muitas situações em sistemas reais. Portanto, quando não há bateria reserva para substituição, devemos maximizar o tempo de operação até a descarga completa. Para calcular o tempo de operação da bateria, nós construímos um modelo ligeiramente diferente, que está representado na Figura 4.15.

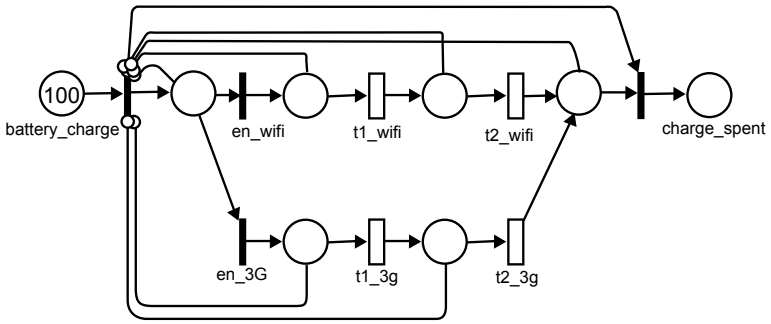


Figura 4.15: Modelo em rede de Petri para descarga da bateria

Este modelo difere do submodelo descrito na Figura 4.14 pela ausência de uma transição de reparo para restaurar os *tokens* de *battery_charge* após uma descarga completa. Este modelo usa a probabilidade estacionária $P\{\#\text{WiFi\_on} = 1\}$ como entrada, calculado através do modelo anterior. Esta probabilidade determina os pesos das transições imediatas *enable_WiFi* e *enable_3g*, relacionadas à taxa de descarga do modelo descrito na Figura 4.15. O peso destas transições é calculado da forma expressa nas Equações (4.14) e (4.15).

$$\text{Weight(enable\_WiFi)} = P\{\#\text{WiFi} = 1\} \tag{4.14}$$

$$\text{Weight(enable\_3G)} = 1 - \text{Weight(enable\_WiFi)} \tag{4.15}$$

A probabilidade P{ #WiFi = 1 } é o parâmetro de entrada do modelo, e é calculado através do modelo da rede de Petri da Figura 4.14.

Uma vez que não há nenhuma transição de reparo para a bateria, este modelo alcança um estado em *deadlock* (ou estado absorvente) quando todos os *tokens* migram de *battery_charge* até *charge_spent*. Este modelo foi avaliado através da ferramenta SHARPE, a qual é capaz de computar o tempo médio para se alcançar o estado absorvente. No modelo da Figura 4.15, este valor corresponde ao tempo médio para descarregar completamente a bateria.

5 Estudos de Caso

Este capítulo apresenta estudos de casos criados com o objetivo de avaliar o impacto das alternativas arquiteturais apresentadas no capítulo anterior sobre a disponibilidade de ambientes de *mobile cloud*. No primeiro estudo de caso, apresentado na Seção 5.1, mostramos os resultados da validação do modelo em RBD da arquitetura base, que é realizada com a ajuda dos resultados obtidos com o modelo de simulação e resultados do experimento de injeção de falhas conduzido. O segundo estudo de caso, apresentado na Seção 5.2, faz uma comparação entre a disponibilidade e *downtime* da arquitetura base de *mobile cloud* com as alternativas arquiteturais propostas com o objetivo de aumentar esta disponibilidade. O terceiro estudo de caso, apresentado na Seção 5.3, mostra uma análise do impacto do mecanismo de redundância no serviço em nuvem proposto na Seção 4.5 sobre a disponibilidade das arquiteturas consideradas. Para investigar os efeitos do uso combinado de múltiplas interfaces de rede sobre o consumo energético no dispositivo móvel e na disponibilidade, apresentamos um quarto estudo de caso na Seção 5.4.

5.1 Validação dos Modelos

A Tabela 5.1 mostra os parâmetros adotados para o cenário base. Os MTTF e MTTR de todos os componentes são assumidos como exponencialmente distribuídos. O MTTF para o *hardware* dos dispositivos *Android* é estimado a partir de dados disponíveis em (Sands & Tseng 2010). O MTTF para o sistema operacional *Android* e aplicações móveis não foram encontrados na literatura, nem publicados pelos fabricantes. Portanto, nós adaptamos valores de (Kim et al. 2009) para tais parâmetros. Usamos o MTTF de sistemas operacionais e aplicações não móveis como uma estimativa para os valores destes componentes no contexto de dispositivos móveis. Para os componentes da nuvem (máquina física, *hypervisor* KVM, aplicação e sistema operacional), nós adotamos valores

de parâmetros usados em (Dantas et al. 2012*a*).

Por questões de conveniência nos experimentos de injeção de falhas, nós agrupamos componentes relacionados do modelo em RBD do cenário base. Em seguida, nós calculamos os MTTF e MTTR para esses grupos de componentes com a ajuda das ferramentas Mercury e SHARPE. A Tabela 5.2 lista os MTTF e MTTR para cada um dos grupos de componentes.

Tabela 5.1: Parâmetros do modelo (em horas)

Componente	MTTF	MTTR
Hardware do disp. móvel (M_HW)	22461,5	1,667
Bateria	9	0,083
SO móvel(M_OS)	1440,9	0,033
App. móvel (M_APP)	336,7	0,0167
Ponto de acesso WiFi	10000	1,667
Sinal WiFi	6	0,078
Hardware do nó da nuvem (N_HW)	8760	1,667
SO do nó da nuvem (N_OS)	2893	0,25
KVM	2990	1
SO da VM (V_SO)	2893	0,25
Aplicação da VM (V_APP)	788	1

Tabela 5.2: Parâmetros dos componentes agrupados

Group	MTTF	MTTR
V_OS+V_APP	619,311	0,840
N_HW+N_OS+KVM	1259,025	0,770
M_HW+Battery	8,996	0,084
M_OS + M_APP	272,925	0,020
WIFI	5,996	0,079

Para o experimento de injeção de falhas, os MTTFs listados na Tabela 5.2 foram reduzidos por um fator de 100 para diminuir o tempo de execução do experimento. O experimento de injeção de falhas durou um total de cinco dias. Após o final do experimento, nós processamos o *log* gerado e avaliamos a disponibilidade $\hat{A}$ aplicando a Equação 5.1.

$$\hat{A} = \frac{\text{Número de amostras “up”}}{\text{Número total de amostras}} \tag{5.1}$$

Para encontrar um intervalo de confiança para a disponibilidade, nós aplicamos o método *Bootstrap* (Efron 1979). O processo é ilustrado na Figura 5.1. A partir do *log*

gerado pelo experimento, extraímos vários *sublogs* aleatórios (*sample log#1, sample log#2, sample log#3,* etc.), e para cada um destes, calculamos sua disponibilidade através da Equação 5.1 (valores *A_b*#1, *A_b*#2, *A_b*#3, etc.). São extraídos um total de 10.000 *sublogs* no processo. Para obter um intervalo de confiança com $\alpha = 0.05$, consultamos os 0.025 e 0.975 quantis dos dados obtidos. Ilustramos esta etapa no histograma da Figura 5.2.

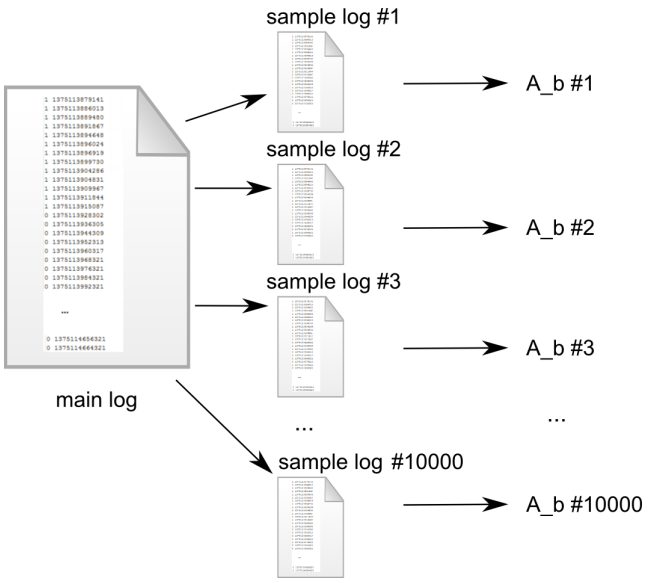


Figura 5.1: *Bootstrap* para cálculo da disponibilidade do experimento

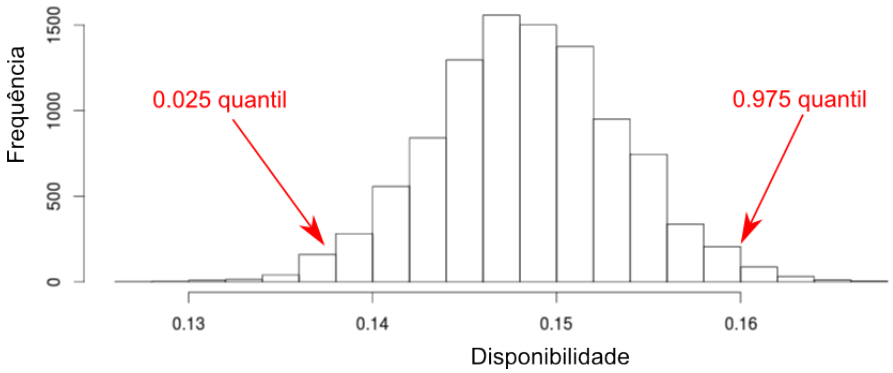


Figura 5.2: Histograma com dados gerados pelo método *bootstrap*

Os resultados para o modelo RBD, modelo de simulação e para o experimento de injeção de falhas estão listados na Figura 5.3. Podemos visualizar nesta figura que o valor obtido pelo modelo analítico permanece dentro dos intervalos de confiança da disponibilidade, obtidos pelo modelo de simulação e pelo experimento de injeção

de falhas. Desta forma, verificamos a equivalência entre os modelos (analítico e de simulação) e o experimento.

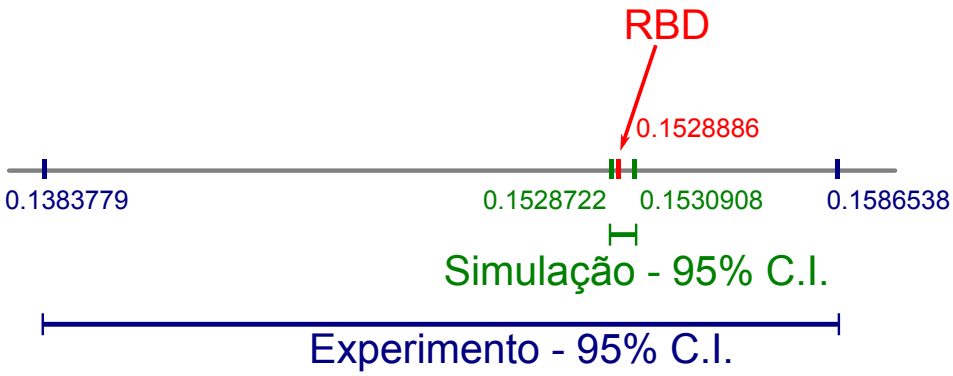


Figura 5.3: Resultados dos modelos e do experimento

Para garantir que nossos experimentos foram consistentes, nós executamos um teste não paramétrico *Mann-Whitney U* entre os MTTFs e MTTRs gerados por simulação, e coletados no experimento. A hipótese nula é que estes valores seguem a mesma distribuição e, portanto, os resultados do experimento e do modelo de simulação estão consistentes entre si. Após executar os testes, obtemos os seguintes *p values*: 0.2787 e 0.3883, para os MTTFs e MTTRs, respectivamente. Desde que estamos considerando um intervalo de confiança de 95% e os *p values* não são inferiores a 0.05, não descartamos a hipótese nula. Portanto, não temos evidência que comprovem que os MTTFs e MTTRs gerados pelo experimento e pela simulação seguem distribuições diferentes e mantemos a premissa de que são consistentes entre si.

5.2 Avaliação das Alternativas Arquiteturais Propostas

Nós avaliamos as arquiteturas apresentadas no Capítulo 4 utilizando os parâmetros da Tabela 5.1 e, adicionalmente, os que estão listados na Tabela 5.3. O tempo médio até a falha de uma estação base 3G pode ser encontrado em (Cooper & Farrell 2007), e o tempo médio até a falha de um cartão de memória *micro SD* foi obtido de (Corporation 2007).

A Tabela 5.4 mostra os resultados do cálculo da disponibilidade estacionária através dos modelos propostos para a arquitetura base e arquiteturas alternativas apresentadas no capítulo anterior. Para visualizar os efeitos da indisponibilidade em cada arquitetura,

Tabela 5.3: Parâmetros adicionais (em horas)

Componente	MTTF	MTTR
(Micro SD) Storage Card	1000000	1
Estação base 3G	83220	12
Sinal 3G	4	0,0777777

exibimos o *downtime* anual na Figura 5.4. Nós verificamos que a adição de uma segunda interface *wireless* promove um acréscimo de 1,25 pontos percentuais na disponibilidade, em comparação à disponibilidade da arquitetura base. Este acréscimo corresponde a uma diferença de cerca de 109,89 horas no *downtime* anual. A arquitetura *cloudlet* causa um aumento ainda mais significativo na disponibilidade, de 1,64 pontos percentuais. Durante o período de um ano, a diferença no *downtime* é de 143,50 horas, comparadas ao *downtime* da arquitetura base.

Tabela 5.4: Comparação da disponibilidade das arquiteturas consideradas

Arquitetura	Disponibilidade
Arquitetura base	0,9740028
Arquitetura store and forward	0,9906802
Múltiplas interfaces de rede	0,9865474
Arquitetura em *cloudlet*	0,9903837

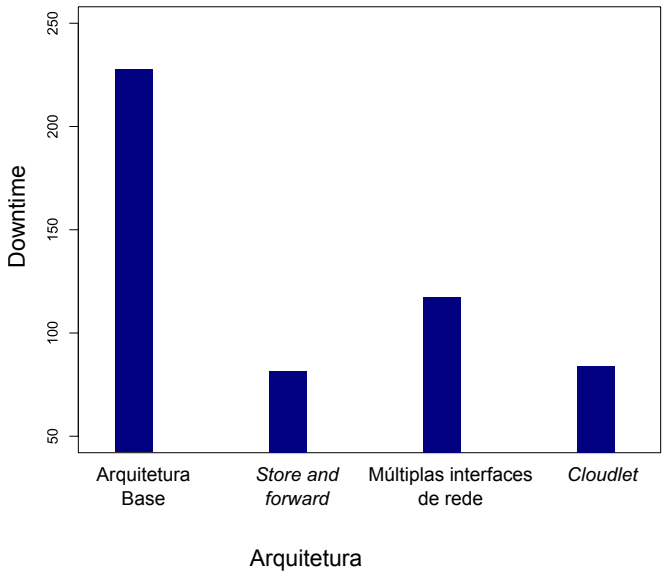


Figura 5.4: *Downtime* anual para cada arquitetura

A disponibilidade da arquitetura *store and forward* é o limite superior para todas as arquiteturas consideradas, uma vez que sua disponibilidade não é afetada pelo ambiente

em nuvem ou pela conectividade *wireless*. Apesar disso, a arquitetura *cloudlet* é capaz de se aproximar deste limite, com uma diferença de apenas $0,02965$ pontos percentuais na disponibilidade. A diferença entre os *downtimes* anuais de ambas as arquiteturas é de apenas $2,60$ horas.

5.3 Avaliação das Arquiteturas com Redundância nos Nós da Nuvem

Nesta seção, avaliamos a disponibilidade das arquiteturas de *mobile cloud*, agora considerando que cada arquitetura incorpora o mecanismo de redundância apresentado na Seção 4.5, que provoca o aumento da disponibilidade do serviço em nuvem. Inicialmente, investigamos o impacto do número de nós de *cluster* sobre a disponibilidade do serviço em nuvem. Este número é representado no modelo da Figura 4.13 pelo parâmetro n. Este parâmetro varia a partir de 2, que é o número mínimo de nós de *cluster* para a infraestrutura considerada (um nó para o servidor *web* e outro para o servidor de banco de dados). Quando $n = 2$, significa que não temos uma máquina extra para implantar VMs quando ocorre alguma falha. A partir de 3 nós, sempre que ocorre uma falha em algum dos dois nós responsáveis por prover o serviço, podemos fazer a implantação da VM em outro nó do *cluster* que esteja à disposição. Estes nós extras também estão sujeitos a falhas, portanto, quanto maior a quantidade n de nós de cluster, maior a probabilidade de termos um nó funcional à disposição. Observe na Tabela 5.5, que o valor da disponibilidade vai aumentando em resposta a cada adição de nó ao *cluster*, mas a partir de $n = 5$, este valor não se altera. Desta forma, determinamos que a fim de atingir o maior nível de disponibilidade, precisamos de no mínimo 5 nós. Porém, a adição de nós extras além deste número não provoca aumento na disponibilidade.

Tabela 5.5: Relação entre a disponibilidade do serviço em nuvem e o número de nós de *cluster*

Número de nós no *cluster*	Disponibilidade
2	0,99563381
3	0,99954153
4	0,99955891
5	0,99955899
6	0,99955899

O gráfico da Figura 5.5 mostra o acréscimo na disponibilidade das arquiteturas avaliadas, obtido através da adição do mecanismo de redundância do serviço em nuvem.

Neste gráfico, a reta horizontal acima das barras verticais equivale à disponibilidade da arquitetura *store and forward*. Uma vez que a disponibilidade desta arquitetura não é afetada por falhas de rede ou da nuvem, este valor é constante e não é afetado pelo mecanismo de redundância. Podemos perceber que o impacto do mecanismo de redundância é maior quando temos apenas uma *cloud* provendo o serviço em nuvem, que é o caso das arquiteturas base e de múltiplas interfaces de rede. Para arquitetura *cloudlet*, o acréscimo da disponibilidade provocado pelo mecanismo de redundância não é tão significativo, conforme apontado pela proximidade entre as barras no gráfico. Essas diferenças também pode ser visualizadas através da Tabela 5.6, que mostra comparação dos *dowmtimes* anuais para as disponibilidades mostradas na Figura 5.5. Observe que a diferença entre os *downtimes* é de aproximadamente 30 horas para as arquiteturas base e de múltiplas interfaces de rede, mas para a *cloudlet*, esta diferença não chega a atingir uma hora.

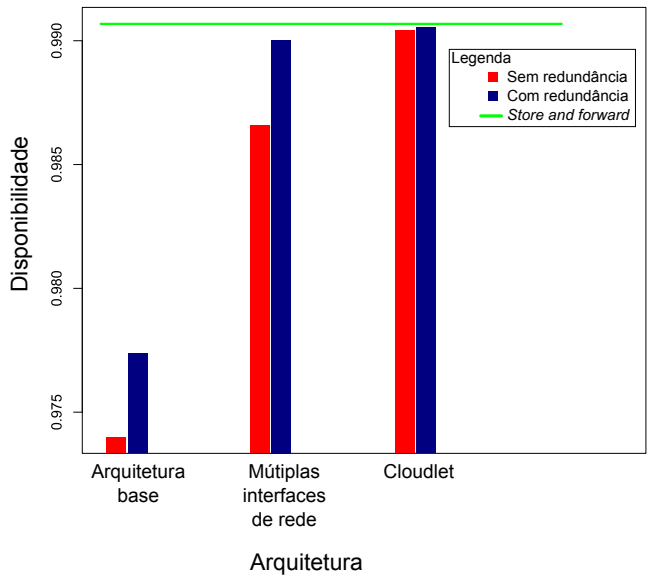


Figura 5.5: Comparação da disponibilidade entre as arquiteturas: sem redundância × com redundância

5.4 Avaliação do Impacto de Interfaces *Wireless* sobre Disponibilidade e Consumo Energético

Em ambientes de *mobile cloud* onde usuários possuem grande mobilidade, um fator que afeta a disponibilidade, do ponto de vista do cliente, é a saída das zonas de cobertura dos

Tabela 5.6: Comparação do *downtime* anual (em horas) entre as arquiteturas: sem redundância × com redundância

Arquitetura	*Downtime* (sem redundância)	*Downtime* (com redundância)	Diferença
Arquitetura base	227,735	197,902	29,833
Múltiplas interfaces de rede	117,845	87,315	30,529
Cloudlet	84,239	83,538	0,7008

access points WiFi e estações base 3G. Dentro da zona de cobertura de um ponto de acesso também existe "pontos cegos", que são regiões nas quais o sinal *wireless* é bloqueado devido a presença de objetos físicos (paredes, portas, etc.). A utilização de múltiplas interfaces de rede poderá provocar um aumento da disponibilidade nestas circunstâncias, uma vez que uma interface poderá ser ativada enquanto a outra estiver indisponível. Contudo, as interfaces 3G e WiFi possuem algumas características conflitantes. WiFi é uma tecnologia de rede *wireless* local (WLAN) capaz de proporcionar velocidades mais altas do que a tecnologia 3G, porém a zona de cobertura dela é menor. Já a tecnologia 3G possui uma largura de banda inferior ao WiFi e, além disso, muitas vezes essa largura de banda diminui ainda mais dependendo da quantidade de dados baixada pelo cliente. Entretanto, o alcance de uma estação base 3G é bem maior que um *access point* WiFi. Adicionalmente, há a questão do consumo energético adicional provocado pelo uso de tais interfaces, conforme apontado na Seção 4.3. Quando temos múltiplas interfaces *wireless* à disposição, devemos priorizar a que apresenta uma menor taxa de consumo de energia.

O objetivo do estudo de caso dessa seção é investigar os efeitos do uso de múltiplas interfaces de rede sobre a disponibilidade e o consumo energético, através dos modelos apresentados na Seção 4.6. A Figura 5.6 exibe um diagrama que ilustra a alternância entre o uso das interfaces *wireless*. Ao longo do uso da aplicação cliente de uma *mobile cloud* que está habilitada para múltiplas interfaces de rede, o usuário irá alternar entre períodos onde ele está conectado pela interface WiFi, períodos onde ele estará conectado via interface 3G, e períodos onde as duas redes estão indisponíveis. Alterando os parâmetros do tempo de bloqueio do modelo da Figura 4.14, podemos alterar a proporção de tempo que ele passa conectado em ambas as interfaces, e verificar o impacto na disponibilidade (através do modelo da Figura 4.14) e no consumo energético (através do modelo da Figura 4.15).

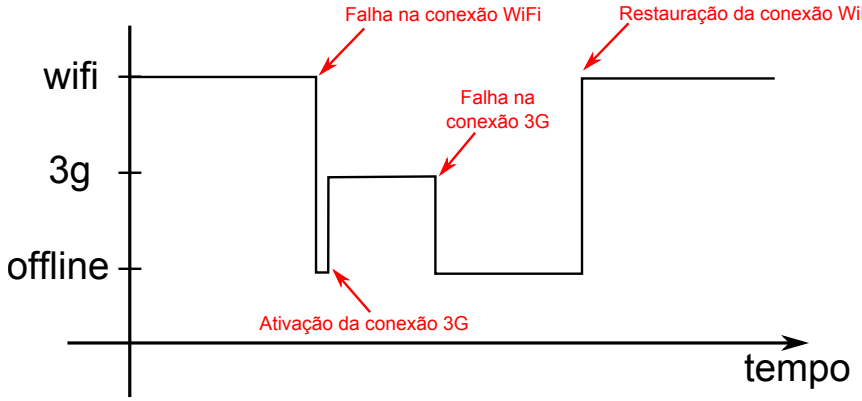


Figura 5.6: Estado da conectividade *wireless* com o uso de múltiplas interfaces de rede

Para este estudo de caso, utilizamos cinco cenários listados abaixo:

- **Cenário número 1** - o cliente possui apenas conexão WiFi disponível;
- **Cenário número 2** - o cliente possui apenas conexão 3G disponível;
- **Cenário número 3** - o cliente possui ambas interfaces habilitadas. O tempo médio de bloqueio dos sinais das duas é igual;
- **Cenário número 4** - o cliente possui ambas interfaces habilitadas. O tempo médio de bloqueio do sinal WiFi é maior que o tempo do sinal 3G;
- **Cenário número 5** - o cliente possui ambas interfaces habilitadas. O tempo médio de bloqueio do sinal 3G é maior que o tempo do sinal WiFi.

A lista dos parâmetros utilizados no modelo em rede de Petri, que são os tempos de disparo das transições que provocam as falhas das redes WiFi e 3G, estão listados na Tabela 5.7.

Os resultados da análise de disponibilidade para os cenários apresentados estão listados na Tabela 5.8. Estes resultados mostram que cenários com apenas uma interface *wireless* habilitada possuem baixa disponibilidade. Os valores para ambos os cenários 1 e 2 são menores que 90%. Estes resultados são explicados pela alta taxa de bloqueio dos sinais *wireless*, definidos pelos tempos médios de disparo das transições $wifi\_b$ e $3g\_b$. É importante notar que a disponibilidade aumenta significativamente quando ambas interfaces são habilitadas (cenários 3, 4 e 5), mas esta medida se mantém abaixo de 99% em todos os cenários. O *downtime* anual para todos os cenários é mostrado na Figura 5.7. Constatamos que o *downtime* diminui mais que 60% quando nós comparamos os cenários

Tabela 5.7: Tempos médios das transições

Transição	Tempo de disparo (horas)	Descrição
ap_fl	10000	Falha no *access point*
ap_rp	12	Reparo do *access point*
bs_fl	83220	Falha na estação base 3G
bs_rp	12	Reparo da estação base 3G
$wifi\_b$	$2,777778$	Entrada em zona de bloqueio WiFi (Cenários 1 e 3)
$wifi\_b$	$4,777778$	Entrada em zona de bloqueio WiFi (Cenário 4)
$wifi\_b$	$0,777778$	Entrada em zona de bloqueio WiFi (Cenário 5)
$wifi\_nb$	$0,2777778$	Saindo de uma zona de bloqueio do WiFi
$3G\_b$	$2,777778$	Entrada em zona de bloqueio 3G (Cenários 2 e 3)
$3G\_b$	$0,777778$	Entrada em zona de bloqueio 3G (Cenário 4)
$3G\_b$	$4,777778$	Entrada em zona de bloqueio 3G (Cenário 5)
$3G\_nb$	$0,2777778$	Saindo de uma zona de bloqueio do 3G

1 ou 2 com os demais cenários. O cenário 3 possui o menor *downtime*, representando um decréscimo de 81.17% em relação ao cenário com o maior *downtime* (cenário 2). De qualquer forma, destacamos que o *downtime* anual permanece na casa das centenas de horas para todos os cenários, uma vez que a disponibilidade não alcança dois noves em nenhum deles.

Tabela 5.8: Resultado da disponibilidade para todos os cenários

Cenário	Descrição	Disponibilidade (%)
1	Apenas WiFi	89,776
2	Apenas 3G	89,392
3	WiFi e 3G com mesma probabilidade de bloqueio	98,003
4	Sinal WiFi mais estável que o 3G	97,479
5	Sinal 3G mais estável que o WiFi	96,787

Baseado nos resultados apresentados, decidimos realizar uma análise de sensibilidade para verificar quanto a disponibilidade estacionária pode ser aumentada através da melhora da probabilidade de bloqueio do sinal WiFi, ou seja, aumentando o tempo médio de disparo da transição $wifi_b$ no modelo em rede de Petri estocástica da Figura 4.14. Este parâmetro foi escolhido porque provoca um grande impacto na disponibilidade do sistema do que o bloqueio do sinal 3G, nos cenários avaliados anteriormente. A probabilidade de bloqueio do sinal WiFi pode ser melhorada em um sistema real através da adição

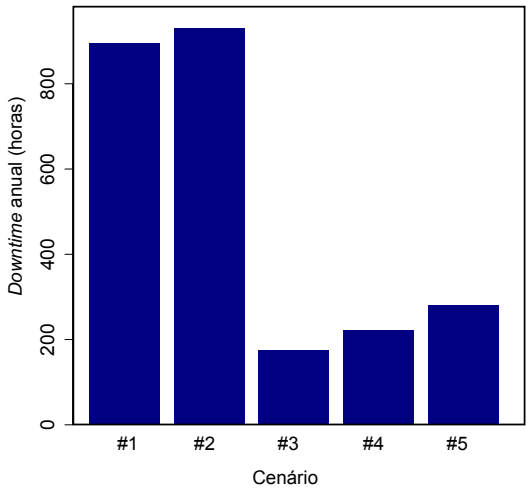


Figura 5.7: *Downtime* anual para os cenários da Tabela 5.8

de *access points*, pela substituição de sua antena por outra de maior ganho, ou pela otimização de sua localização. Estas ações poderão expandir a zona de cobertura do WiFi. Outra ação que pode reduzir esta probabilidade é a limitação da mobilidade do usuário, que pode ser difícil de ser alcançada. Nós variamos o tempo médio da transição $wifi_b$ de 1 até 10 horas, fixamos os outros parâmetros do cenário, e usamos o modelo para calcular a disponibilidade de cada ponto. A Figura 5.8 descreve os pontos resultantes desta análise de sensibilidade. A curva cheia representa o modelo de regressão linear (Draper et al. 1966) obtido através do *fitting* dos pontos do gráfico. Este modelo de regressão é representado pela Equação 5.2.

$$A(t) = 0,99289 - 0,02879 \cdot (1/t^{0,8}) \tag{5.2}$$

O máximo valor para esta função é de $\lim_{t\to\infty} A(t) = 0,99289$, indicando que este é o valor máximo para a disponibilidade que pode ser alcançado através do aumento do tempo médio de bloqueio do sinal WiFi, mantendo os outros parâmetros inalterados. Portanto, outras características do sistema devem ser investigadas e melhoradas a fim de prover uma disponibilidade mais alta. O valor de $0,99289$ para a disponibilidade estacionária corresponde a um *downtime* anual de $62,2$ horas, que é 64% menor que o *downtime* do cenário que apresenta a maior disponibilidade (cenário 3), o que corresponde a uma melhora significativa. Devido a natureza exponencial da função ajustada, há um número infinito de valores para $wifi\_b$ que correspondem à máxima disponibilidade. Contudo, considerando um sistema que deve prover, por exemplo, pelo menos dois noves de disponibilidade (ou seja, $A(t) \geq 0,99$), o valor mínimo para $wifi\_b$ é de $17,69$ horas.

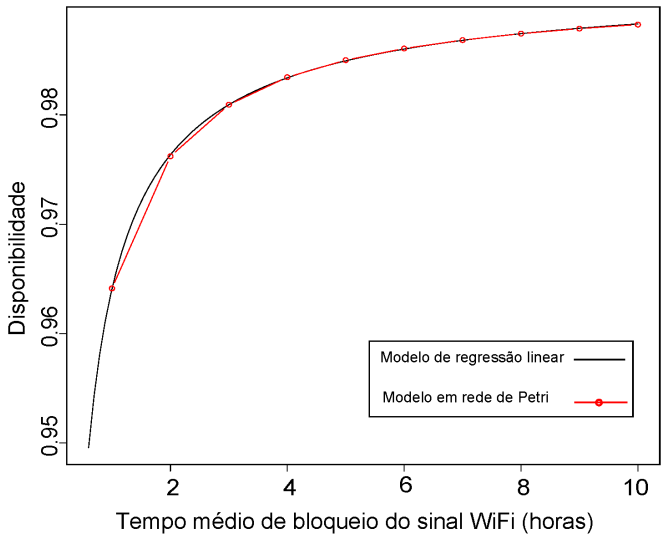


Figura 5.8: Impacto da probabilidade de bloqueio do sinal WiFi na probabilidade do sistema

O gráfico na Figura 5.9 mostra o tempo médio para a descarga da bateria, estimado através do modelo da Figura 4.15, e considerando os mesmos cenários da Tabela 5.8. A Figura 5.9 mostra que a interface *wireless* pode ter um impacto significativo no tempo médio de descarga da bateria. A diferença entre os cenários #1 e #2 (somente WiFi versus somente 3G) é de $1,609$ horas, quase $13,26\%$ do tempo total do cenário #1. Quando ambas as interfaces estão ativadas e a interface WiFi tem maior prioridade (cenário #4), o tempo médio de descarga é maior que o do cenário #2, até mesmo em uma situação onde a interface WiFi desconecta com maior frequência (cenário #5).

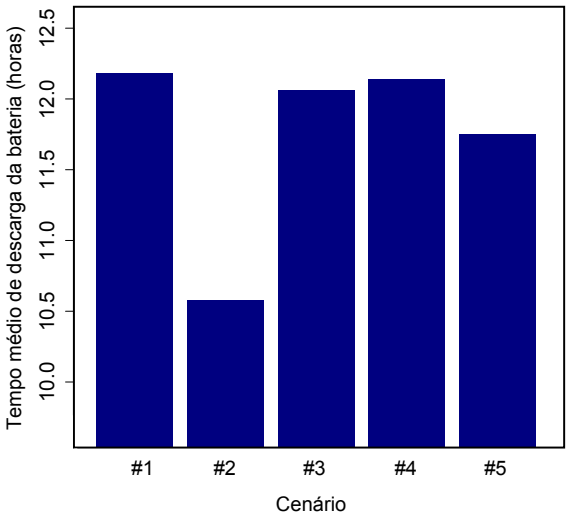


Figura 5.9: Tempo médio de descarga da bateria

5.5 Considerações Finais

Os estudos de caso deste capítulo apresentaram uma discussão sobre aspectos relacionados à disponibilidade de ambientes de *mobile cloud computing*. Os resultados obtidos forneceram *insights* que proporcionam uma maior compreensão dos fatores que provocam impactos negativos e positivos sobre a disponibilidade de tais ambientes.

Em primeiro lugar, vimos que a alternativa arquitetural *store and forward* é a que oferece a maior disponibilidade em aplicativos móveis. Esta arquitetura permite que clientes móveis de um serviço em nuvem continuem utilizando a aplicação mesmo com a indisponibilidade do serviço em nuvem ou de uma conexão *wireless*. Aplicativos como o *Evernote*[1] e *Any Do*[2] funcionam desta forma, permitindo que o usuário crie e gerencie anotações e tarefas, mesmo com o aplicativo *offline*. Quando o usuário consegue estabelecer uma conexão com a Internet, estes aplicativos realizam a sincronização dos dados locais com a nuvem. Entretanto, esta solução não é aplicável em todas as situações. Por exemplo, aplicativos como *Foursquare*[3] , *Whatsapp*[4] e *Facebook*[5] ficam completamente inutilizados devido a ausência de uma conexão com a nuvem.

Uma conclusão importante alcançada neste trabalho, foi que a arquitetura *cloudlet* oferece um aumento significativo na disponibilidade, próximo ao proporcionado pela arquitetura *store and forward*. Embora o propósito de uma *cloudlet* seja melhorar o tempo de resposta de aplicações móveis em nuvem, a melhoria na disponibilidade se configura como um grande benefício adicional provido por esta arquitetura.

Também concluímos que a utilização de múltiplas interfaces de rede também promovem um aumento da disponibilidade em comparação com a arquitetura básica de *mobile cloud*, que utiliza apenas uma interface de rede. Entretanto, devemos priorizar o uso de redes WLAN de alta velocidade sobre as redes 3G, pois, além da menor latência, conexões WiFi também provocam um impacto menor sobre o consumo energético de aplicações móveis do que conexões 3G.

[1] Evernote, http://www.evernote.com/
[2] Any.do, http://www.any.do/
[3] Foursquare, http://www.foursquare.com/
[4] WhatsApp, http://www.whatsapp.com/
[5] Facebook, http://www.facebook.com/

6

Conclusões e Trabalhos Futuros

O crescimento acelerado do mercado de dispositivos como *tablets* e *spartphones* está mudando o panorama da Internet. Atualmente, cerca de 10% do tráfego da Internet é gerado por dispositivos móveis (Hollister 2012), e este percentual só tende a aumentar. Combinado com a evolução das tecnologias de telecomunicação sem fio, isto define um cenário promissor para o desenvolvimento de aplicações móveis, que atuam como pontos de entrada para serviços em nuvem. Infelizmente, as restrições desta plataforma computacional - mais notavelmente a carga limitada da bateria do dispositivo e a instabilidade das conexões sem fio - podem levar à insatisfação do usuário com os serviços oferecidos.

Neste contexto, nós realizamos neste trabalho um estudo de avaliação de disponibilidade em ambientes de *mobile cloud*. Nós investigamos três diferentes cenários que tem como premissa a melhoria da disponibilidade de uma arquitetura básica de *mobile cloud computing*: arquitetura *store and forward*, múltiplas interfaces de rede e uma arquitetura baseada em *cloudlet*. Para avaliar os cenários, nós criamos um modelo hierárquico baseado em RBD e CTMC, que foi validado através de um experimento de injeção de falhas e um modelo de simulação.

Considerando que a arquitetura *store and forward* habilita o sistema a continuar sua execução mesmo sem conexão com a nuvem, esta arquitetura determina o limite superior da disponibilidade entre os cenários estudados. Contudo, nossos resultados mostraram que a arquitetura *cloudlet* foi capaz de melhorar a disponibilidade significativamente, chegando a se aproximar da disponibilidade oferecida pela arquitetura *store and forward*.

Também investigamos mecanismos de redundância no lado da nuvem, utilizando mais nós de cluster para implantação de VMs. Para avaliar este mecanismo de redundância, utilizamos um modelo em rede de Petri. Os resultados obtidos a partir deste modelo indicam uma melhora significativa na disponibilidade para os cenários base e com múltiplas interfaces de rede. Contudo, quando consideramos o uso deste mecanismo em

conjunto com a arquitetura baseada em *cloudlet*, a melhoria é insignificante.

Por último, fizemos um estudo da relação entre uso de interfaces *wireless* por aplicações móveis e o consumo energético, e sua relação com a disponibilidade do sistema. Para isso, combinamos o uso de modelos RBD e redes de Petri. Considerando diferentes cenários de utilização, os resultados mostraram que a disponibilidade mais alta ocorre quando temos sinal WiFi e 3G em boas condições. Considerando sinais com diferentes condições de estabilidade, a disponibilidade é mais alta quando o sinal WiFi é mais estável do que o sinal 3G do que o cenário oposto. Uma análise de sensibilidade por regressão nos permitiu determinar o nível máximo de melhoria de disponibilidade que pode ser alcançada variando o parâmetro do tempo médio de falha do sinal WiFi. Também detectamos que o tempo médio de operação da bateria é bastante semelhante para a maioria dos cenários avaliados, exceto pelo cenário onde apenas o sinal 3G está habilitado, o que possui o maior índice de descarga da bateria, seguido pelo cenário onde o sinal 3G é mais estável que o sinal WiFi.

6.1 Contribuições

Apresentamos as principais contribuições deste trabalho na listagem a seguir:

- Um *framework* para a avaliação de dependabilidade em aplicações móveis na nuvem através de experimentação foi desenvolvido. Este *framework* fica como legado e poderá ser utilizado em diferentes trabalhos futuros;
- Uma API de simulação baseada no NS-2, para a avaliação de disponibilidade de sistemas. Esta API permite ajudar a validação de modelos, e a modelagem de cenários cujos componentes apresentem tempos médios de falha e reparo que seguem distribuições diferentes da exponencial;
- Uma metodologia de avaliação de disponibilidade para aplicações móveis que utilizam recursos de uma nuvem computacional. Esta metodologia faz uso de modelagem hierárquica com modelos em RBD e redes de Petri, experimento de injeção de falhas e modelos de simulação, e tem como objetivo avaliar a efetividade de soluções arquiteturais na melhoria de disponibilidade para *mobile clouds*;
- Uma discussão detalhada sobre diferentes arquiteturas de aplicações móveis em nuvem, e o impacto destas soluções sobre a disponibilidade do sistema. Vimos que a arquitetura *cloudlet* proposta por (Satyanarayanan et al. 2009) oferece um alto

nível de disponibilidade, além do benefício principal que é a redução do tempo de resposta do serviço;

- Publicação de artigo em conferência internacional (Qualis B):

  Danilo Oliveira, Jean Araujo, Rubens Matos and Paulo Maciel. Availability and Energy Consumption Analysis of Mobile Cloud Environments. In: Proceedings of the 2013 IEEE International Conference on Systems, Man, and Cybernetics (IEEE SMC 2013). October 13-16, 2013 - Manchester, United Kingdom.

6.2 Trabalhos Futuros

A seguir, listamos as extensões do trabalho atual que serão realizadas em trabalhos futuros:

- Desenvolver o mecanismo de redundância que foi apresentado e modelado na seção 4.5, e testar sua efetividade através de experimentação com injeção de falhas;
- Realizar testes de consumo energético mais acurados, em especial testes com conexão 3G, uma vez que os valores de descarga 3G foram obtidos de forma indireta;
- Estudar outros atributos de dependabilidade em ambientes de *mobile cloud*, além da disponibilidade;
- Realizar estudos de desempenho e performabilidade nas arquiteturas consideradas de *mobile cloud*;

Referências Bibliográficas

Acker, E. V., Weber, T. S. & Cechin, S. L. (2010), Injeção de falhas para validar aplicações em ambientes móveis, *in* 'Workshop de Testes e Tolerância a Falhas', Vol. 11, pp. 61–74.

AmazonS3 (2013), 'Amazon simple storage service (amazon s3)'. Disponível em: http://aws.amazon.com/s3/.

Android (2013), 'Philosophy and goals', Android Open Source Project. Disponível em: *http://source.android.com*.

Archer, J., Boehme, A., Cullinane, D., Puhlmann, N., Kurtz, P. & Reavis, J. (2012), 'Security guidance for critical areas of mobile computing'.
URL: *https://cloudsecurityalliance.org/download/security-guidance-for-critical-areas-of-mobile-computing/*

Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.-C., Laprie, J.-C., Martins, E. & Powell, D. (1990), 'Fault injection for dependability validation: A methodology and some applications', *Software Engineering, IEEE Transactions on* **16**(2), 166–182.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. et al. (2010), 'A view of cloud computing', *Communications of the ACM* **53**(4), 50–58.

Avizienis, A., Laprie, J.-C., Randell, B. et al. (2001), *Fundamental concepts of dependability*, University of Newcastle upon Tyne, Computing Science.

Balasubramanian, N., Balasubramanian, A. & Venkataramani, A. (2009), Energy consumption in mobile phones: a measurement study and implications for network applications, *in* 'Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference', ACM, pp. 280–293.

Berthomieu, B. & Diaz, M. (1991), 'Modeling and verification of time dependent systems using time petri nets', *Software Engineering, IEEE Transactions on* **17**(3), 259–273.

Bolch, G., Greiner, S., de Meer, H. & Trivedi, K. S. (1998), *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*, Wiley-Interscience, New York.

Bolch, G., Greiner, S., de Meer, H. & Trivedi, K. S. (2006), *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*, John Wiley & Sons.

Boyle, C. (2014), 'Smartphone users worldwide will total 1.75 billion in 2014'. Disponível em http://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536.

Callou, G., Maciel, P., Tutsch, D., Ferreira, J., Araújo, J. & Souza, R. (2013), 'Estimating sustainability impact of high dependable data centers: a comparative study between brazilian and us energy mixes', *Computing* **95**(12), 1137–1170.

Carolan, J., Gaede, S., Baty, J., Brunette, G., Licht, A., Remmell, J., Tucker, L. & Weise, J. (2009), 'Introduction to cloud computing architecture', *SUN Microsystems Inc., pp.–1-40* .

Chen, D., Kintala, C., Garg, S. & Trivedi, K. S. (2003), Dependability enhancement for ieee 802.11 wireless lan with redundancy techniques, *in* 'Proceedings of the International Conference on Dependable Systems and Networks', pp. 521–528.

Chiola, G., Marsan, M. A., Balbo, G. & Conte, G. (1993), 'Generalized stochastic petri nets: A definition at the net level and its implications', *Software Engineering, IEEE Transactions on* **19**(2), 89–107.

Chun, B.-G. & Maniatis, P. (2009), Augmented smartphone applications through clone cloud execution, *in* 'Proceedings of the 12th conference on Hot topics in operating systems', pp. 8–8.

Ciardo, G., German, R. & Lindemann, C. (1994), 'A characterization of the stochastic process underlying a stochastic petri net', *Software Engineering, IEEE Transactions on* **20**(7), 506–515.

Cinque, M., Cotroneo, D., Kalbarczyk, Z. & Iyer, R. K. (2007), How do mobile phones fail? a failure data analysis of symbian os smart phones, *in* 'Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on', IEEE, pp. 585–594.

Cinque, M., Cotroneo, D. & Testa, A. (2012), A logging framework for the on-line failure analysis of android smart phones, *in* 'Proceedings of the 1st European Workshop on AppRoaches to MObiquiTous Resilience', ACM, p. 2.

Cooper, T. & Farrell, R. (2007), Value-chain engineering of a tower-top cellular base station system, *in* 'Vehicular Technology Conference, 2007. VTC2007-Spring. IEEE 65th', IEEE.

Corporation, S. (2007), 'Sandisk sd card product family'.
URL: *http://media.digikey.com/pdf/Data Sheets/M-Systems Inc PDFs/SD Card Prod Family OEM Manual.pdf*

Courtes, L., Hamouda, O., Kaaniche, M., Killijian, M.-O. & Powell, D. (2007), Dependability evaluation of cooperative backup strategies for mobile devices, *in* 'Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on', IEEE, pp. 139–146.

Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R. & Bahl, P. (2010), Maui: making smartphones last longer with code offload, *in* 'Proceedings of the 8th international conference on Mobile systems, applications, and services', ACM, pp. 49–62.

Dantas, J., Matos, R., Araujo, J. & Maciel, P. (2012*a*), An availability model for eucalyptus platform: An analysis of warm-standy replication mechanism, *in* 'Proceedings of the 2012 IEEE International Conference on Systems, Man, and Cybernetics (IEEE SMC 2012)'.

Dantas, J., Matos, R., Araujo, J. & Maciel, P. (2012*b*), 'Models for dependability analysis of cloud computing architectures for eucalyptus platform', *International Transactions on Systems Science and Applications* **8**, 13–25.

Dinh, H. T., Lee, C., Niyato, D. & Wang, P. (2011), 'A survey of mobile cloud computing: architecture, applications, and approaches', *Wireless Communications and Mobile Computing* .

Doukas, C., Pliakas, T. & Maglogiannis, I. (2010), Mobile healthcare information management utilizing cloud computing and android os, *in* 'Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE', IEEE, pp. 1037–1040.

Draper, N. R., Smith, H. & Pownell, E. (1966), *Applied regression analysis*, Vol. 3, Wiley New York.

Drebes, R. J. (2005), FIRMAMENT: Um Módulo de Injeção de Falhas de Comunicação para Linux, Master's thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre.

Dropbox (2013), 'Dropbox'. Disponível em: https://www.dropbox.com.

Efron, B. (1979), 'Bootstrap methods: another look at the jackknife', *The annals of Statistics* **7**(1), 1–26.

Eucalyptus (2013), 'Eucalyptus - the open source cloud platform', Eucalyptus Systems. Disponível em: http://open.eucalyptus.com/.

Friedman, R., Kogan, A. & Krivolapov, Y. (2013), 'On power and throughput tradeoffs of wifi and bluetooth in smartphones', *Mobile Computing, IEEE Transactions on* **12**(7), 1363–1376.

German, R. (1996), A concept for the modular description of stochastic petri nets (extended abstract, *in* 'Proc. 3rd Int. Workshop on Performability Modeling of Computer and Communication Systems', pp. 20–24.

Giurgiu, I., Riva, O., Juric, D., Krivulev, I. & Alonso, G. (2009), Calling the cloud: enabling mobile phones as interfaces to cloud applications, *in* 'Middleware 2009', Springer, pp. 83–102.

Glassfish (2013), 'World's first java ee 7 application server'. Disponível em: https://glassfish.java.net/.

H. Schneider, V. L. & Schell, R. (2004), Introduction to mobile application architectures, *in* 'Mobile Applications:Architecture,Design, andDevelopment', Pearson Information IT.

Hoang, D. B. & Chen, L. (2010), Mobile cloud for assistive healthcare (mocash), *in* 'Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific', IEEE, pp. 325–332.

Hof, R. D. (2006), 'Jeff bezos' risky bet'. Disponível em: http://www.businessweek.com/stories/2006-11-12/jeff-bezos-risky-bet.

Hollister, S. (2012), 'Mobile devices account for nearly 10 percent of internet traffic, according to statcounter'. Disponível em: http://www.theverge.com/2012/5/11/3012957/mobile-devices-account-for-nearly-10-percent-of-internet-traffic.

Hsueh, M. C., Tsai, T. K. & Iyer, R. K. (1997), 'Fault injection techniques and tools', *Computer* **30**(4), 75–82.

Ingalls, R. G. (2008), Introduction to simulation, *in* 'Proceedings of the 40th Conference on Winter Simulation', Winter Simulation Conference, pp. 17–26.

Kalic, G., Bojic, I. & Kusek, M. (2012), Energy consumption in android phones when using wireless communication technologies, *in* 'MIPRO, 2012 Proceedings of the 35th International Convention', IEEE, pp. 754–759.

Kartson, D., Balbo, G., Donatelli, S., Franceschinis, G. & Conte, G. (1994), *Modelling with generalized stochastic Petri nets*, John Wiley & Sons, Inc.

Killijian, M.-O., Powell, D., Banâtre, M., Couderc, P. & Roudier, Y. (2004), Collaborative backup for dependable mobile applications, *in* 'Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing', ACM, pp. 146–149.

Kim, D. S., Ghosh, R. & Trivedi, K. S. (2010), 'A hierarchical model for reliability analysis of sensor networks', *Pacific Rim International Symposium on Dependable Computing, IEEE* **0**, 247–248.

Kim, D. S., Machida, F. & Trivedi, K. S. (2009), Availability modeling and analysis of a virtualized system, *in* 'Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on', IEEE, pp. 365–371.

Klein, A., Mannweiler, C., Schneider, J. & Schotten, H. D. (2010), Access schemes for mobile cloud computing, *in* 'Mobile Data Management (MDM), 2010 Eleventh International Conference on', IEEE, pp. 387–392.

Kleinrock, L. (1975), *Theory, volume 1, Queueing systems*, Wiley-interscience.

Koopman, S. (2012), 'The mobile cloud computing market will generate 45 billion dollars in revenues by 2016 | asd reports'. Disponível em https://www.asdreports.com/news.asp?prid=200.

Kumar, K. & Lu, Y.-H. (2010), 'Cloud computing for mobile users: Can offloading computation save energy?', *Computer* **43**(4), 51–56.

Lee, K., Lee, J., Yi, Y., Rhee, I. & Chong, S. (2010), Mobile data offloading: how much can wifi deliver?, *in* 'Proceedings of the 6th International COnference', ACM, p. 26.

Lin, W., Chiu, D. M. & Lee, Y. (2004), Erasure code replication revisited, *in* 'Peer-to-Peer Computing, 2004. Proceedings. Proceedings. Fourth International Conference on', IEEE, pp. 90–97.

Logothetis, D. & Trivedi, K. (1995), Time-dependent behavior of redundant systems with deterministic repair, *in* 'Computations with Markov Chains', Springer, pp. 135–150.

Maciel, P. R. M., Trivedi, K. S., Matias, R. & Kim, D. S. (2011), *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*, Information Science Reference.

Maciel, P. R. M., Trivedi, K. S., Matias, R. & Kim, D. S. (2012), 'Dependability modeling', pp. 53 –97.

Malhotra, M. (1994), 'Power-hierarchy of dependability model types', *IEEE Trans. on Reliability* **43**(2), 493–502.

Malhotra, M. & Trivedi, K. (1993), A methodology for formal expression of hierarchy in model solution, *in* 'Petri Nets and Performance Models, 1993. Proceedings., 5th International Workshop on', pp. 258–267.

Markov, A. A. (1906), 'Extension of the law of large numbers to dependent quantities', *Izv. Fiz.-Matem. Obsch. Kazan Univ.(2nd Ser)* **15**, 135–156.

Marwah, M., Maciel, P., Shah, A., Sharma, R., Christian, T., Almeida, V., Araújo, C., Souza, E., Callou, G., Silva, B. et al. (2010), 'Quantifying the sustainability impact of data center availability', *ACM SIGMETRICS Performance Evaluation Review* **37**(4), 64–68.

Matos, R. (2011), An automated approach for systems performance and dependability improvement through sensitivity analysis of markov chains, Master's thesis, Universidade Federal de Pernambuco, Recife.

Meeker, M. (2012), 'Internet trends'. Disponível em: http://www.businessinsider.com/mary-meeker-2012-internet-trends-year-end-update-2012-12.

Meenaghan, P. & Delaney, D. (2004), An Introduction to NS, Nam and OTcl scripting - Network Simulator, Technical report, National University of Ireland, Maynooth, Maynooth, Co. Kildare, Ireland.

Mell, P. & Grance, T. (2011), 'The nist definition of cloud computing (draft)', *NIST special publication* **800**, 145.

Menasce, D. A., Almeida, V. A., Dowdy, L. W. & Dowdy, L. (2004), *Performance by design: computer capacity planning by example*, Prentice Hall Professional.

Nicol, D. M., Sanders, W. H. & Trivedi, K. S. (2004), 'Model-based evaluation: from dependability to security', *Dependable and Secure Computing, IEEE Transactions on* **1**(1), 48–65.

OpenNebula (2013), 'Opennebula - the open source solution for data center virtualization', OpenNebula. Available in: http://opennebula.org/.

OpenStack (2013), 'Openstack open source cloud computing software', OpenStack. Disponível: http://www.openstack.org/.

Pandey, S. & Nepal, S. (2012), Modeling availability in clouds for mobile computing, *in* 'Mobile Services (MS), 2012 IEEE First International Conference on', IEEE, pp. 80–87.

Peng, J., Zhang, X., Lei, Z., Zhang, B., Zhang, W. & Li, Q. (2009), Comparison of several cloud computing platforms, *in* 'Proc. of 2nd Int. Symp. on Information Science and Engineering (ISISE)', IEEE Press, Shanghai, pp. 23–27.

PostgreSQL (2013), 'The world's most advanced open source database'. Disponível em: http://www.postgresql.org/.

Qi, H. & Gani, A. (2012), Research on mobile cloud computing: Review, trend and perspectives, *in* 'Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on', IEEE, pp. 195–202.

Ra, M.-R., Paek, J., Sharma, A. B., Govindan, R., Krieger, M. H. & Neely, M. J. (2010), Energy-delay tradeoffs in smartphone applications, *in* 'Proceedings of the

8th international conference on Mobile systems, applications, and services', ACM, pp. 255–270.

Rausand, M. & Høyland, A. (2003), *System reliability theory: models, statistical methods, and applications*, Vol. 396, Wiley-Interscience.

Reisig, W. & Rozenberg, G. (1998), *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, Vol. 149, Springer.

Sahner, R. A., Trivedi, K. S. & Puliafito, A. (1996), *Performance and reliability analysis of computer systems: an example-based approach using the SHARPE software package*, Kluwer Academic Publishers.

Sands, A. & Tseng, V. (2010), 'Cell phone comparison study nov 10'. Disponível em http://www.squaretrade.com/cell-phone-comparison-study-nov-10.

Sareen, P. (2013), 'Cloud computing: Types, architecture, applications, concerns, virtualization and role of it governance in cloud', *International Journal* **3**(3).

Satyanarayanan, M. (1996), Fundamental challenges in mobile computing, *in* 'Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing', ACM, pp. 1–7.

Satyanarayanan, M., Bahl, P., Caceres, R. & Davies, N. (2009), 'The case for vm-based cloudlets in mobile computing', *Pervasive Computing, IEEE* **8**(4), 14–23.

Silva, B. (2013), 'Mercury tool'. Disponível em: https://sites.google.com/site/mercurytooldownload/.

Silva, B., Callou, G., Tavares, E., Maciel, P., Figueiredo, J., Sousa, E., Araujo, C., Magnani, F. & Neves, F. (2013), 'Astro: An integrated environment for dependability and sustainability evaluation', *Sustainable Computing: Informatics and Systems* **3**(1), 1 – 17.

Tang, W.-T., Hu, C.-M. & Hsu, C.-Y. (2010), A mobile phone based homecare management system on the cloud, *in* 'Biomedical Engineering and Informatics (BMEI), 2010 3rd International Conference on', Vol. 6, IEEE, pp. 2442–2445.

Trivedi, K. S. (2010), 'Sharpe portal'. Disponível em: http://sharpe.pratt.duke.edu/.

Trivedi, K., Vasireddy, R., Trindade, D., Nathan, S. & Castro, R. (2006), Modeling high availability, *in* 'Dependable Computing, 2006. PRDC '06. 12th Pacific Rim International Symposium on', pp. 154–164.

Truong, H.-L. & Dustdar, S. (2009), On analyzing and specifying concerns for data as a service, *in* 'Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific', IEEE, pp. 87–94.

Vaquero, L. M., Rodero-Merino, L., Caceres, J. & Lindner, M. (2008), 'A break in the clouds: towards a cloud definition', *ACM SIGCOMM Computer Communication Review* **39**(1), 50–55.

Varshney, U. (2007), 'Pervasive healthcare and wireless health monitoring', *Mobile Networks and Applications* **12**(2-3), 113–127.

Watson, J.F., I. & Desrochers, A. (1991), 'Applying generalized stochastic petri nets to manufacturing systems containing nonexponential transition functions', *Systems, Man and Cybernetics, IEEE Transactions on* **21**(5), 1008–1017.

Wetherall, D. (2009), 'Otcl'. Disponível em: http://otcl-tclcl.sourceforge.net/otcl/.

Wolfgang Reisig (2013), *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*, Springer. 230 pages; ISBN 978-3-642-33277-7.

Zhang, Q., Cheng, L. & Boutaba, R. (2010), 'Cloud computing: state-of-the-art and research challenges', *Journal of Internet Services and Applications* **1**(1), 7–18.

Zhang, X., Jeong, S., Kunjithapatham, A. & Gibbs, S. (2010), Towards an elastic application model for augmenting computing capabilities of mobile platforms, *in* 'Mobile wireless middleware, operating systems, and applications', Springer, pp. 161–174.

Zhao, M. & Figueiredo, R. J. (2007), Experimental study of virtual machine migration in support of reservation of cluster resources, *in* 'Proceedings of the 2nd international workshop on Virtualization technology in distributed computing', ACM, p. 5.

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

Zimmermann, A. (2013), 'Timenet'. Disponível em: http://www.tu-ilmenau.de/sse/timenet/.

Zimmermann, A., Freiheit, J., German, R. & Hommel, G. (2000), Petri net modelling and performability evaluation with timenet 3.0, *in* 'Computer Performance Evaluation. Modelling Techniques and Tools', Springer, pp. 188–202.

Zuberek, W. (1991), 'Timed petri nets definitions, properties, and applications', *Microelectronics Reliability* **31**(4), 627–644.

Appendices

A

Script de injeção de falhas

Listagem A.1: *Script Perl* para injeção de falhas

```
1  #!/usr/bin/perl -w
2
3  use Time::HiRes qw/ time sleep /;
4
5  $command_fail = "virsh␣--connect␣qemu:///system␣shutdown␣vm1";
6  $command_repair = "virsh␣--connect␣qemu:///system␣start␣vm1";
7
8  $component_name = "V_SO+V_APP";
9  $fail_time = 22295.2;
10 $repair_time = 3022.25;
11
12 sub exp_sleep{
13     $x =  - $_[0] * log( rand() );
14     print "Sleep:␣$x\n";
15     sleep( $x );
16 }
17
18 sub append_to_log{
19         open(LOGFILE, '>>log.txt');
20         $timestamp = int(time() * 1000);
21         print LOGFILE @_, $timestamp, "\n";
22         close(LOGFILE);
23 }
24
25 append_to_log( "start␣" );
26
27 while(1){
```

```
28      exp_sleep( $fail_time );
29      append_to_log( "fail␣$component_name␣" );
30      system($command_fail);
31      exp_sleep( $repair_time );
32      append_to_log( "repair␣$component_name␣" );
33      system($command_repair);
34  }
```

B

API de simulação

Listagem B.1: API de simulação em OTcl

```
1  Class Component
2
3  Component instproc init {n sim} {
4      $self instvar name
5      $self instvar operational
6      $self instvar fail_timestamp
7      $self instvar repair_timestamp
8      $self instvar n_failures
9      $self instvar n_repairs
10     $self instvar simulator
11     $self instvar MTTR
12     $self instvar MTTF
13     $self instvar activated
14     $self instvar log
15     $self instvar debug
16
17     set simulator $sim
18     set fail_timestamp 0
19     set repair_timestamp 0
20     set operational 1
21     set MTTR 0
22     set MTTF 0
23     set name $n
24     set activated 1
25     set log 0
26     set debug 0
27 }
```

```
28
29 Component instproc availability {} {
30         $self instvar MTTF MTTR
31         return [expr $MTTF / ($MTTR + $MTTF)]
32 }
33
34 Component instproc repair {} {
35         $self instvar availability
36         $self instvar repair_timestamp
37         $self instvar fail_timestamp
38         $self instvar simulator
39         $self instvar MTTR
40         $self instvar debug
41         $self instvar name
42
43         incr n_repairs
44
45         set repair_timestamp [$simulator now]
46
47         set TTR [expr $repair_timestamp - $fail_timestamp]
48         set MTTR [expr $MTTR + $TTR]
49
50         $self setStatus 1
51
52         if {$debug} {
53
54                 #puts "repair $name [$simulator now]"
55         }
56 }
57
58 Component instproc fail {} {
59         $self instvar availability
60         $self instvar fail_timestamp
61         $self instvar repair_timestamp
62         $self instvar simulator
63         $self instvar MTTF
64         $self instvar debug
65         $self instvar name
66
67         incr n_failures
68
69         set fail_timestamp [$simulator now]
70
```

```
71          set TBF [expr [$simulator now] - $fail_timestamp]
72          set TTF [expr $fail_timestamp - $repair_timestamp]
73
74          set MTTF [expr $MTTF + $TTF]
75
76          $self setStatus 0
77
78          if {$debug} {
79                  #puts "fail $name [$simulator now]"
80          }
81  }
82
83  Component instproc setStatus {status} {
84          $self instvar name
85      $self instvar operational
86      $self instvar listener
87      $self instvar parent
88
89      set operational $status
90
91          if {[info exists parent]} {
92                  $parent notify $self
93          }
94  }
95
96  Component instproc evaluate {} {
97      $self instvar operational
98      return $operational
99  }
100
101 Class CompoundComponent -superclass Component
102
103 CompoundComponent instproc init {n sim} {
104     $self next $n $sim
105     $self instvar components
106
107     set components {}
108 }
109
110 CompoundComponent instproc addComponent {comp} {
111     $self instvar components
112     $comp set parent $self
113
```

```
114     lappend components $comp
115 }
116
117 Class SeriesComponent -superclass CompoundComponent
118
119 SeriesComponent instproc init {n sim} {
120     $self next $n $sim
121 }
122
123 SeriesComponent instproc evaluate {} {
124     set result 1
125
126     $self instvar components
127     $self instvar operational
128
129     foreach comp $components {
130         set comp_eval [$comp evaluate]
131         set result [expr $result && $comp_eval]
132     }
133
134     return $result
135 }
136
137 SeriesComponent instproc notify {comp} {
138         $self instvar operational
139
140         set status_ [$self evaluate]
141
142         if {$status_ != $operational} {
143                 if {$status_} {
144                     #$self activateComponents 1
145                         $self repair
146                 } else {
147                     #$self activateComponents 0
148                         $self fail
149                 }
150         }
151 }
152
153 SeriesComponent instproc startAll {} {
154         $self instvar components
155
156     foreach c $components {
```

```
157                 $c start
158     }
159 }
160
161 SeriesComponent instproc activateComponents {status} {
162         $self instvar components
163
164     foreach c $components {
165                 $c set activated $status
166     }
167 }
168
169
170 Class ExponentialComponent -superclass Component
171
172 ExponentialComponent instproc init {name sim fR rR} {
173     $self next $name $sim
174     $self instvar simulator
175     $self instvar failureRate
176     $self instvar repairRate
177     $self instvar generatorFailure
178     $self instvar generatorRepair
179
180     set simulator $sim
181     set failureRate $fR
182     set repairRate $rR
183
184     set generatorFailure [new RandomVariable/Exponential]
185         $generatorFailure set avg_ $fR
186
187         set generatorRepair [new RandomVariable/Exponential]
188         $generatorRepair set avg_ $rR
189 }
190
191 ExponentialComponent instproc start {} {
192         $self instvar simulator
193         $self instvar generatorFailure
194         $self instvar failEvent
195
196         set t [expr [$generatorFailure value]    + [$simulator now]]
197         set failEvent [$simulator at $t "$self generateFailure"]
198 }
199
```

```
200 ExponentialComponent instproc stop {} {
201         $self instvar simulator
202         $self instvar failEvent
203 
204         $simulator cancel failEvent
205 }
206 
207 ExponentialComponent instproc generateFailure {} {
208 
209         $self instvar simulator
210         $self instvar generatorRepair
211         $self instvar generatorFailure
212         $self instvar name
213         $self instvar failEvent
214         $self instvar activated
215 
216         if {$activated} {
217             $self fail
218             set t [expr [$generatorRepair value] + [$simulator now]]
219             set failEvent [$simulator at $t "$self␣generateRepair"]
220         } else {
221             set t [expr [$generatorFailure value] + [$simulator now]]
222             set failEvent [$simulator at $t "$self␣generateFailure"]
223         }
224 }
225 
226 ExponentialComponent instproc generateRepair {} {
227         $self instvar simulator
228         $self instvar generatorFailure
229         $self instvar name
230 
231         $self repair
232 
233         set t [expr [$generatorFailure value] + [$simulator now]]
234         $simulator at $t "$self␣generateFailure"
235 }
```

Listagem B.2: *Script* de simulação para a arquitetura básica

```
1 source components.tcl
2 
3 set ns [new Simulator]
```

```
4
5  set FACTOR [lindex $argv 0]
6
7  set vso_vapp01 [new ExponentialComponent vso_vapp01
8     $ns [expr 619.311057/$FACTOR] 0.839513719]
9  set nhw_nso_kvm01 [new ExponentialComponent nhw_nso_kvm01
10    $ns [expr 1259.02508/$FACTOR] 0.769546048]
11 set node01 [new SeriesComponent node01 $ns]
12 $node01 addComponent $vso_vapp01
13 $node01 addComponent $nhw_nso_kvm01
14
15 set vso_vapp02 [new ExponentialComponent vso_vapp02
16    $ns [expr 619.311057/$FACTOR] 0.839513719]
17 set nhw_nso_kvm02 [new ExponentialComponent nhw_nso_kvm02
18    $ns [expr 1259.02508/$FACTOR] 0.769546048]
19 set node02 [new SeriesComponent node02 $ns]
20
21 $node02 addComponent $vso_vapp02
22 $node02 addComponent $nhw_nso_kvm02
23
24 set cloud [new SeriesComponent cloud $ns]
25 $cloud addComponent $node01
26 $cloud addComponent $node02
27
28 set mhw_battery [new ExponentialComponent mhw_battery
29    $ns [expr 8.99639527/$FACTOR] 0.0839736488]
30 set mso_mapp [new ExponentialComponent mso_mapp
31    $ns [expr 272.924747/$FACTOR] 0.0198238595]
32 set mobile_device [new SeriesComponent mobile_device $ns]
33 $mobile_device addComponent $mhw_battery
34 $mobile_device addComponent $mso_mapp
35
36 set wifi [new ExponentialComponent wifi
37    $ns [expr 5.99640216/$FACTOR] 0.0787433764]
38 set series01 [new SeriesComponent series01 $ns]
39
40 $series01 addComponent $cloud
41 $series01 addComponent $mobile_device
42 $series01 addComponent $wifi
43
44 $node01 startAll
45 $node02 startAll
46 $mobile_device startAll
```

```
47 $wifi start
48
49 proc finish {} {
50         global ns series01
51         puts [$series01 availability]
52         $ns halt
53 }
54
55 proc ping {} {
56         global ns series01
57
58         puts "[$series01␣set␣operational]␣[$ns␣now]"
59
60         set t [expr (5/3600.0) + [$ns now]]
61         $ns at $t "ping"
62 }
63
64 $series01 set debug 1
65
66 $defaultRNG seed [string rang [clock clicks -milliseconds] 10 end]
67
68 $ns at 50000 "finish"
69
70 $ns run
```