UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Simulação Distribuída em Cloud Computing

Rodrigo da Silva Vaz

Itajubá, Novembro de 2015

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Rodrigo da Silva Vaz

Simulação Distribuída em Cloud Computing

Dissertação submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciência e Tecnologia da Computação

Área de Concentração: Sistemas de Computação

Orientador: Prof. Dr. Edmilson Marmo Moreira
Coorientador: Prof. Dr. Otávio Augusto Salgado Carpinteiro

Novembro de 2015
Itajubá - MG

Agradecimentos

Ao Prof. Dr. Edmilson Marmo Moreira, pela orientação, apoio e incentivo, sem os quais esse trabalho não seria realizado.

Aos meus pais Roberto e Nazaré pelo incentivo moral e monetário.

Aos meus irmãos Lilian e Samuel por me apoiarem e acreditarem em mim.

À minha namorada Ariane pelo incentivo e apoio.

À minha sogra Neuza pelo apoio.

Ao meu amigo Elton por me ajudar e me ouvir nas horas de desânimo.

Aos meus amigos Tabuti, Danilo e Rafael pelo apoio e amizade.

Aos meus professores Ivan e Christiane.

Aos professores de Engenharia da Computação da UNIFEI.

Aos professores do Mestrado em Ciência e Tecnologia da Computação da UNIFEI.

Resumo

A computação em nuvem é um recurso que está sendo cada vez mais utilizado, seja por programadores ou clientes de um *software*. Além dos serviços de *software* que estão migrando para essa plataforma, um programador também pode aproveitar os recursos de um provedor utilizando máquinas virtuais remotas, que podem ser alugadas e depois utilizadas por uma interface de navegador ou linha de comando. Como a simulação é utilizada em muitas áreas científicas, é importante adaptá-la à nuvem. Em simulações mais demoradas, é costume separar partes das simulações em processos independentes utilizando programação paralela, finalidade para qual protocolos como o *Time Warp* e o *Rollback Solidário* foram desenvolvidos. Infelizmente, a computação em nuvem tem a desvantagem de possuir diversas máquinas físicas e estrutura de rede sendo utilizados por diversos clientes diferentes com requerimentos diferentes, gerando uma grande instabilidade na rede, processamento e memória das máquinas virtuais. Além disso, com a adaptação de protocolos otimistas aos ambientes de nuvem pública, surge o problema do aumento da quantidade de *rollbacks*, que faz com que as simulações levem mais tempo para serem executadas. Como é demonstrado, já existem trabalhos que propuseram otimizações para o protocolo *Time Warp*, além de alguns tratando sobre o preço e a vantagem do ganho de desempenho e economia, quando se utiliza um ambiente de nuvem para aplicações científicas. Neste contexto, este trabalho apresenta uma solução, que possibilita que um programador consiga analisar, através de um simulador de computação em nuvem, todo o histórico de envio e recebimento de mensagens, assim como os atrasos envolvidos em relação à rede. Com esta solução, é possível propor otimizações nos protocolos de simulação, melhorando o seu desempenho e confiabilidade durante a execução neste tipo de arquitetura computacional. O simulador escolhido foi o *NetworkCloudSim*, onde foram feitas alterações para adaptá-lo a executar *threads* e para que o desempenho de rede fosse mais instável, se aproximando do caso real de serviços públicos de nuvem.

Abstract

Cloud computing is a resource that is being used more and more, by programmers and software clients. Despite the software services that are migrating to this platform, a programmer can also take advantage of a provider resources by using remote virtual machines, that can be rented and later used by a web interface or command line. As simulations are used on many scientific fields, it's important to adapt them to cloud computing. In the most time consuming simulations, is usual to separate parts of simulations in independent processes using parallel programming, goal that made protocols like Time Warp and *Rollback Solidário* being developed. Unfortunately, cloud computing have the disadvantage of having several physical machines and network structures being used for many different clients with different requirements, making the network very unstable as processing and memory of virtual machines as well. Besides, with the optimistic protocols for distributed simulation to public cloud computing, come the increasing of rollbacks, that make the simulations more time consuming. As will be shown, there are already some works proposing optimizations to Time Warp Protocol and others about the price and advantages of the performance increase and money saving when one uses cloud computing for scientific purposes. On that context, this work presents a solution that makes a programmer able to analyse, using a cloud computing simulator, the messages sends and receives history, as well as the network packets delays. With that solution, it's possible to propose optimizations on simulations protocols, improving their performance and reliability during it's execution on this kind of computing architecture. The chosen simulator was NetworkCloudSim, which was modified and adapted for threads execution, and for making the network performance instable, getting close to the real public cloud computing performance cases.

Sumário

Lista de Figuras

Lista de Tabelas

Glossário

API	*Aplication Programing Interface*
CV	*Coeficiente de variação*
E/S	*Entrada/Saída*
GVT	*Global Virtual Time*
HLA	*High-Level Architecture*
HPC	*High Performance Computing*
IaaS	*Infrastructure as a Service*
LP	*Logical Process*
LVT	*Local Virtual Time*
MIPS	*Millions of Instructions per Second*
PaaS	*Platform as a service*
RTI	*Run-Time Infrastructure*
RTT	*Round-trip time*
SaaS	*Software as a Service*
SO	*Sistema Operacional*
vCPU	*virtual CPU*
VM	*Virtual Machine*
VMM	*Virtual Machine Monitor*

1 Introdução

Computação em nuvem é um modelo - geralmente de negócios - que permite o acesso a recursos de um ambiente computacional, via rede - como a internet - ou, pela definição completa: *"é um modelo que permite acesso via rede, sob demanda, conveniente e onipresente a um conjunto compartilhado de recursos computacionais configuráveis que pode ser rapidamente provido e liberado com o mínimo esforço de gerenciamento ou interação com o provedor de serviço"* (MELL; GRANCE, 2009). Por conta dessa característica, esse modelo redefiniu rapidamente a forma com que as pessoas enxergam a computação, pois permite a desenvolvedores e pesquisadores o acesso a máquinas e redes virtuais através de uma interface simples e normalmente com formas de pagamento que levam em consideração a quantidade utilizada dos recursos disponíveis.

A simulação distribuída é uma tecnologia que permite que múltiplos processos executem em computadores distintos conectados por uma rede (PIENTA; FUJIMOTO, 2013). Essa área pode se beneficiar bastante do modelo de computação em nuvem, pois a possibilidade de se conectar máquinas virtuais através de uma rede virtual e obter acesso ao sistema operacional dessas máquinas, permite que essas simulações sejam executadas. No entanto, para implementar essas simulações, utilizam-se protocolos, como o *Time Warp* (JEFFERSON, 1985) e o *Rollback Solidário* (MOREIRA, 2005), que são o foco desse trabalho e possuem como característica a identificação de uma mensagem que não está de acordo com o tempo da simulação - fato que pode ocorrer, por exemplo, por atrasos de pacotes na rede - e o retorno dos processos da simulação a um estado consistente - que possui valores das variáveis de um processo em um determinado momento - em que a execução pode ser retomada a fim de evitar o erro que poderia ser causado pelo atraso. Uma característica dos serviços públicos de computação em nuvem, é o fato de possuírem diversos clientes executando tarefas diferentes e utilizando ou não a rede de forma intensiva em certos momentos, tornando o desempenho inconsistente, como será analisado posteriormente nesse trabalho, aumentando os atrasos de pacotes e consequentemente as mensagens da simulação, causando um aumento no número de *rollbacks*, fato observado por Fujimoto, Malik e

Park (2010). Pensando nesse problema, e como pode ser complicado pagar um serviço de nuvem para se executar testes de protocolo, que dependem de momentos de instabilidade imprevisíveis, propôs-se nesse trabalho, desenvolver um ambiente que permita o controle da instabilidade, com a finalidade de que o programador consiga implementar melhorias no protocolo de simulação distribuída, para resolver problemas específicos.

O objetivo desse trabalho é alterar o *framework* de simulação de computação em nuvem chamado *NetworkCloudSim* (GARG; BUYYA, 2011), para permitir a implementação de protocolos otimistas de simulação distribuída e então utilizá-lo como base para analisar o comportamento das mensagens na ocorrência de atrasos aleatórios, baseados em dados obtidos de um serviço público de computação em nuvem, possibilitando que melhorias sejam propostas aos protocolos existentes. A necessidade de se utilizar um modelo simulado de computação em nuvem surge, em primeiro lugar, pois, os serviços são pagos sob demanda. Em segundo lugar, é mais interessante analisar o comportamento passo a passo e de forma controlada, ou seja, os atrasos podem ser modificados e seus valores podem ser visualizados, ao contrário do ambiente real em que não se sabe antecipadamente quando um atraso vai ocorrer e qual será o tamanho do atraso. Para se compreender melhor o trabalho proposto, serão tratados tópicos sobre simulação distribuída (Capítulo 2) e computação em nuvem (Capítulo 3) e também diversos trabalhos que apresentaram formas de se executar uma simulação distribuída em computação em nuvem (Capítulo 4). Após esse estudo, será apresentado o *framework* e as alterações elaboradas neste trabalho, além do seu funcionamento (Capítulo 5). Por fim, tem-se a conclusão (Capítulo 6) e uma amostra da configuração do simulador utilizado (Apêndice A).

2 Simulação Distribuída

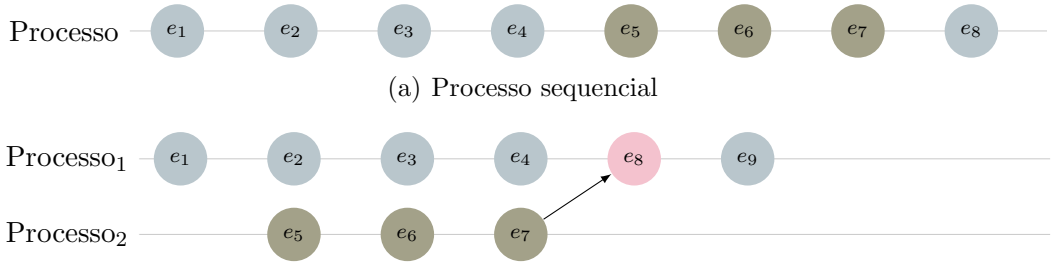
É considerada simulação, além da definição apresentada no capítulo 1, uma imitação de uma operação ou processo que ocorre ao longo do tempo no mundo real (BANKS et al., 2009), portanto, o trânsito em uma cidade ou filas de caixa de supermercado são sistemas e podem ser simulados; mas o que é simulado de um sistema não são todos os elementos da cena, mas apenas os que são de interesse, ou seja, os que implicam diretamente no comportamento a ser estudado. Essa abstração do sistema real é chamada de modelo.

Um modelo - ou abstração - de um sistema, além de uma representação, é também uma simplificação suficientemente detalhada para que a situação do mundo real seja analisada com o máximo de precisão. Sendo assim, é possível pensar em um modelo, por exemplo uma miniatura de um carro para a análise em laboratório da sua aerodinâmica; um modelo matemático consistindo em um conjunto de fórmulas e tabelas; um modelo simbólico como um diagrama; ou um programa representando a abstração.

Existem diversos tipos ou categorias de simulação, como a contínua, de Monte Carlo e a simulação de eventos discretos. Uma simulação contínua é aquela que simula um sistema em que as variáveis de estado mudam continuamente com o tempo (BANKS et al., 2009). Na simulação de Monte Carlo, números aleatórios são gerados e usados para simular eventos aleatórios nos modelos (SINGH, 2009). O modelo de simulação computacional aqui tratado é a simulação de eventos discretos, que consiste em modelar o sistema pela representação de como ele evolui no tempo, com as variáveis de estado se modificando em instantes separados (LAW; KELTON, 2000).

A simulação distribuída tratada nesse trabalho, considera os casos em que o tamanho do modelo inviabiliza a execução da simulação de forma sequencial, possibilitando que eventos que não possuem relação causal entre si sejam executados em paralelo em processos distintos, como observado na Figura 1(a), que ilustra um caso em que os eventos $\{5, 6, 7\}$ são responsáveis por processar um dado que será usado em $\{8\}$ e os eventos $\{1, 2, 3\}$ processam o dado utilizado em $\{4\}$, podendo redistribuí-los em dois processos distintos,

como observado na Figura 1(b).



(a) Processo sequencial

(b) Divisão em dois processos

Figura 1: Separando os eventos em dois processos

A simulação distribuída não é tão trivial quanto se pode imaginar inicialmente, pois os processos são executados individualmente, em máquinas diferentes gerando problemas como a dificuldade de se identificar se uma nova mensagem chegou em tempo de ser processada ou se atrasou o suficiente para que um evento que a precedia fosse executado sem o dado nela contido. Tais problemas, são causados não só pelos atrasos inerentes à estrutura de rede, mas também pela ausência de um relógio global de referência, dificultando a sincronização dos processos. Como tentativas de resolver esse problema, surgiram outras formas de sincronização. Essas abordagens podem ser conservativas ou otimistas. A sincronização conservativa procura evitar problemas na sincronização, enquanto que a otimista detecta e recupera esses erros quando eles ocorrem, como os protocolos *Time Warp* (JEFFERSON, 1985) e *Rollback Solidário* (MOREIRA, 2005). Nesse trabalho, foi implementado um protocolo otimista, pois explora melhor o paralelismo do modelo estudado.

2.1 Sincronização dos processos

Como mencionado, um grande problema em uma simulação distribuída é a sincronização. Foram desenvolvidas formas diferentes de abordar o problema. Os mecanismos conservativos e otimistas consideram que os processos lógicos (PLs) se comunicam trocando mensagens contendo um *timestamp* T_s, que é um tempo lógico de recebimento, garantindo que os eventos ocorram de acordo com a ordem imposta pelos relógios. Os protocolos otimistas, permitem a ocorrência de problemas, se preocupando apenas em como detectá-los e resolvê-los, abordagem que garante um grau maior de paralelismo.

Nos mecanismos otimistas, se uma mensagem recebida com um T_s menor do que o esperado - chamada de "desgarrada" (*straggler*) - é detectada, o algoritmo retorna a um estado em que não havia inconsistências, ação que é chamada de *rollback*. Para efetuar esse retorno, os estados de cada processo devem ser salvos em determinados instantes. O

conjunto dos estados salvos pelos processos da simulação é chamado de estado global, que é consistente quando todos os eventos de recebimento que foram salvos possuem os seus correspondentes de envio também salvos. Um subconjunto do estado global é chamado de corte. Como exemplo de um caso em que o salvamento de estados não gerou um estado global consistente, pode ser observado na figura 2, onde o evento de envio e_0^2 não foi salvo, mas foi salvo o recebimento e_2^3, sendo que e_p^t representa um evento e no tempo lógico t do processo p. Essa inconsistência surge caso seja necessário retornar o processo p_0 para um estado anterior ao evento e_0^2 e o processo p_2 continue após e_2^3.

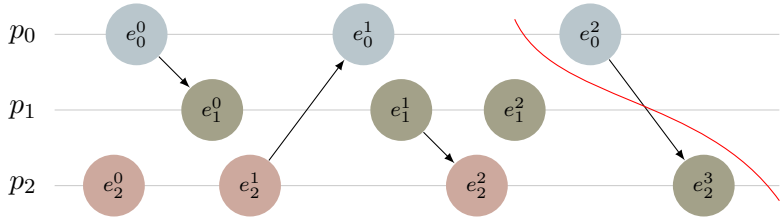


Figura 2: Um corte inconsistente (em vermelho) em um programa distribuído

Os protocolos otimistas *Time Warp* e *Rollback Solidário* apresentam formas diferentes de retornar a um **estado global consistente**, sendo que o primeiro retorna pelo envio de mensagens para anular quaisquer mensagens que foram enviadas após o momento do estado global que se deseja retornar e o segundo utiliza o conceito de *checkpoint* global consistente (BABAOGLU; MARZULLO, 1993), garantindo que os estados salvos possam ser utilizados de forma consistente.

2.1.1 Time Warp

Esse protocolo é baseado em **tempo virtual** ou **tempo de simulação**. Nele, a mensagem desgarrada é detectada quando contém um T_s menor do que o *timestamp* do processo que a recebe. Nesta situação, o *rollback* é realizado desfazendo todos os efeitos dos eventos que foram processados entre o tempo no qual o processo recebeu a mensagem e o tempo em que ele deseja retornar, que são todos os eventos com T_s maior do que o da mensagem desgarrada (BAUER et al., 2015).

No *Time Warp* as mensagens contêm, além dos dados que se deseja transmitir, **identificação do remetente**, **identificação do receptor**, **tempo virtual do envio**, **tempo virtual de recebimento** e **sinal**. O **tempo virtual** (JEFFERSON, 1985) é uma variável inteira, mantida em cada processo e que é incrementada após cada evento, sendo que o tempo virtual do envio é o valor dessa variável do processo remetente no momento do envio e o tempo virtual de recebimento é o valor do tempo no qual se espera que essa

mensagem seja processada no processo receptor. O sinal da mensagem é utilizado para indicar se a mensagem recebida é comum ou uma **antimensagem**, um tipo especial de mensagem que é enviada para anular os efeitos de uma mensagem enviada anteriormente.

Quando uma mensagem é enviada, ela contém um sinal positivo (+) no seu campo de sinal, sendo que nesse momento, o processo coloca uma cópia dessa mensagem em sua fila de saída, mas com o sinal negativo (-). Uma mensagem com o sinal negativo é chamada de antimensagem. Quando a mensagem positiva chega ao receptor, ela é inserida à fila de entrada, que é mantida para o processo poder retornar ao tempo de uma das mensagens. A fila de saída, que contém as antimensagens, é mantida para o processo identificar quais mensagens ele precisa desfazer ao voltar no tempo. Para desfazer a mensagem, o processo simplesmente transmite a antimensagem, que ao ser recebida pode causar um outro *rollback* no receptor, anular uma mensagem recebida mas ainda não processada ou aguardar até que a sua correspondente positiva chegue e seja assim anulada.

Resumidamente, o *Time Warp* permite que os processos troquem mensagens normalmente, apenas checando o seu tempo virtual de recebimento e o seu sinal, no momento do processamento da mensagem. Se o tempo virtual de recebimento for maior ou igual ao tempo virtual do processo, ela deve ser processada normalmente, se for menor, é escolhido um estado para retornar e as antimensagens com tempo virtual de envio entre o tempo atual e o que se deseja retornar são enviadas para os respectivos remetentes. Se a mensagem recebida for uma antimensagem, as ações mencionadas anteriormente podem ser tomadas, dependendo do instante em que ela foi recebida, sendo que em um caso de *rollback* causado por uma antimensagem, pode provocar novos *rollbacks* em outros processos, até que se atinja o estado global consistente.

2.1.2 Rollback Solidário

No protocolo *Rollback Solidário*, ao receber uma mensagem desgarrada, o processo lógico escolhe o estado que deve ser recuperado. Depois da escolha do estado, o sistema busca os estados que são consistentes com o estado escolhido e entre si, utilizando a teoria de *checkpoints globais consistentes* (MOREIRA, 2005). De maneira geral, os protocolos *Time Warp* e *Rollback Solidário* se comportam de forma semelhante, sendo que a grande diferença está na hora de realizar o *rollback*.

Para salvar os estados, o *Rollback Solidário* pode usar uma estratégia síncrona, ou seja, quando um processo decide realizar um *checkpoint*, o protocolo, através de um dos processos lógicos, coordena todos os processos para a obtenção de um *checkpoint global*

consistente. Outra estratégia que pode ser utilizada é a semi-síncrona, que seria analisar os casos que geram *checkpoints globais inconsistentes* e tentar evitá-los.

Um *checkpoint* é um estado salvo. Um conjunto de *checkpoints*, um de cada processo, é um *checkpoint* global. Um *checkpoint* global é consistente se os *checkpoints* dos processos não são causalmente dependentes entre si. Para evitar a dependência causal, se um *checkpoint* vai restaurar um instante em que uma mensagem será enviada, o *checkpoint* do processo alvo da mensagem deve restaurar o instante antes dessa mensagem ser recebida ou ambos devem restaurar estados após essa mensagem ser enviada e recebida.

Para sincronizar o salvamento dos estados, os processos devem realizar os *checkpoints* de forma sincronizada, mantendo o *checkpoint global* sempre consistente, ou de uma maneira semi-síncrona, que busca evitar um caminho em um intervalo de *checkpoints*, ou seja, os eventos entre dois *checkpoints* de dois ou mais processos, chamado de caminho-Z (KSHEMKALYANI; SINGHAL, 2008). O caminho-Z pode ser causal, quando há uma dependência causal entre dois ou mais eventos ou não causal, quando uma mensagem é enviada antes da que foi encaminhada anteriormente ser recebida.

O funcionamento desse protocolo, como mencionado, é semelhante ao *Time Warp*, mas com diferenças na abordagem do *rollback*. Dessa forma, a simulação também ocorre normalmente, até que uma mensagem desgarrada seja detectada, mas, ao contrário do protocolo apresentado anteriormente, ao invés de enviar antimensagens que podem gerar novos envios de antimensagens em outros processos, esse protocolo vai salvar estados globais consistentes, seja de forma síncrona ou semi-síncrona, permitindo que um *rollback* ocorra de forma imediata, simplesmente informando aos outros processos que o retorno é necessário.

2.2 Considerações finais

Esse capítulo apresentou um breve estudo sobre simulação distribuída e algumas definições sobre simulação, mas com o foco em protocolos otimistas de sincronização, abordando o *Time Warp* e o *Rollback Solidário*. É importante destacar que existem diversas melhorias para o *Time Warp*, como a adaptação a ambientes de máquinas com processadores de muitos núcleos (CHEN et al., 2011), *clusters beowulf* com máquinas de muitos núcleos (DICKMAN; GUPTA; WILSEY, 2013), sistemas com acesso à memória não uniforme (PELLEGRINI; QUAGLIA, 2015), melhoria no desempenho pelo estudo e adaptação a memórias transacionais (DIEGUES; ROMANO, 2015) e ajustes nas frequências dos núcleos para

acelerar as simulações (CHILD; WILSEY, 2012a, 2012b).

O protocolo *Time Warp* é a base usada para análise de novos protocolos, como o *Rollback Solidário*. Embora não seja muito utilizado na sua forma original, existem muitos trabalhos buscando melhorias no algoritmo, como a proposta de Tay e Teo (2000) ou do Vee e Hsu (2002), além das tentativas de adaptá-lo aos ambientes em nuvem, que serão discutidas no capítulo 4. A principal diferença entre os protocolos aqui estudados é que no *Rollback Solidário*, por não haver envio de antimensagem, não há o risco do *rollback* em um processo gerar diversos *rollbacks* em cascata.

Embora a simulacão distribuída permita a otimização do tempo de execução, talvez o usuário não tenha um *cluster* à disposição ou talvez seja necessário que a simulação seja observada à distância. Levando em conta tais problemas, o projeto pode ser inviabilizado, pois nem sempre a otimização do tempo é vantajosa, sendo necessário o estudo de uma nova plataforma para a implementação de simulações distribuídas. Para resolver esse tipo de demanda, existe a computação em nuvem, que será discutida no próximo capítulo.

3 Cloud Computing

O termo computação em nuvem (*Cloud Computing*), considerado por alguns como um termo introduzido pela mídia, diz respeito a um subconjunto da computação em grade (*Grid Computing*) voltado ao uso de recursos especiais compartilhados (AYMERICH; FENU; SURCIS, 2008). Segundo Mell e Grance (2009), computação em nuvem *"é um modelo que permite acesso via rede, sob demanda, conveniente e onipresente a um conjunto compartilhado de recursos computacionais configuráveis que pode ser rapidamente provido e liberado com o mínimo esforço de gerenciamento ou interação com o provedor de serviço"*.

Os serviços que podem ser oferecidos são, geralmente, *software* (*Software as a Service – SaaS*), plataforma (*Platform as a Service – PaaS*) e infraestrutura (*Infrastructure as a Service – IaaS*). O SaaS oferece um *software* que é executado remotamente na infraestrutura de nuvem. O PaaS permite ao usuário do serviço criar um *software* dentro da infraestrutura de nuvem. Por último, o IaaS é uma solução completa com acesso ao processamento, armazenamento e rede – entre as máquinas virtuais – tornando o ambiente uma simulação de uma máquina real. Como comparação, em cada tipo de serviço, são oferecidos mais privilégios (acesso a um *software* para execução – acesso a *softwares* e bibliotecas para programação – acesso a uma máquina virtual completa) (GABRIELSOON et al., 2010).

Em um serviço de computação em nuvem, é preciso que haja uma nuvem (*Cloud*), ou seja, serviços de infraestrutura, geralmente virtualizados, oferecidos pelas organizações e uma tecnologia de nuvem (*Cloud Technology*) que diz respeito ao *software* responsável por oferecer o serviço de execução, armazenamento ou comunicação (EKANAYAKE et al., 2009). Em resumo, computação em nuvem é um paradigma que coloca a computação como serviço, permitindo a um usuário utilizar capacidades do *hardware* contratado (como armazenamento ou processamento) remotamente, pagando-se por esse serviço ou não. Alguns exemplos de serviços de computação em nuvem são o Amazon EC2 (2015), GoGrid (2015) e ElasticHosts (2015) e exemplos de tecnologias de nuvem são Google MapReduce (DEAN; GHEMAWAT, 2004), Apache Hadoop (2015) e Dryad (2015).

Uma das tecnologias mais importantes, que fez a computação em nuvem possível, é a virtualização. Para oferecer serviços como os mostrados na Figura 3, é necessário o uso de um *hardware* virtualizado, pois uma máquina virtual (VM) isola as aplicações do *hardware* e de outras máquinas virtuais e permite a personalização de acordo com as necessidades de cada usuário (BUYYA et al., 2009). Serviços como o Amazon EC2 e *softwares* como o OpenStack permitem a requisição remota da criação de uma VM no servidor, que acessa dispositivos de entrada e saída (E/S) suportados por um hipervisor - chamado também de monitor de máquina virtual, VMM. Para uma estrutura de SaaS ou PaaS, ainda é necessário um *software* executando dentro das VMs; mas esse não é o foco desse trabalho.

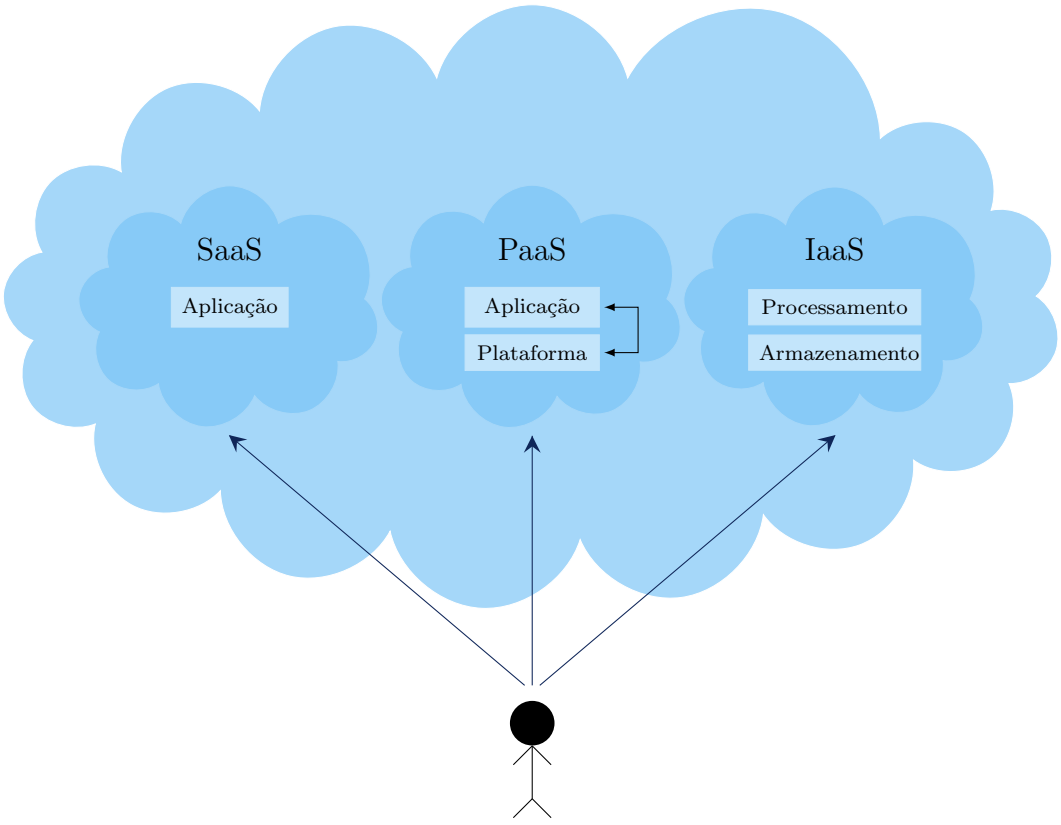


Figura 3: Serviços na computação em nuvem

3.1 Arquitetura

Em um primeiro contato com um serviço de nuvem, observa-se que é possível gerenciar VMs utilizando uma interface *web* ou comandos remotos, que criam, suspendem, migram, entre outras operações, VMs em uma estrutura física remota com vários computadores ligados por uma rede. Também se sabe que um serviço de nuvem pode disponibilizar a execução remota de programas do provedor ou prover um ambiente para que o usuário

crie seus programas ou ainda entregar um acesso completo a uma ou várias VMs. Para entender como tudo isso funciona, é preciso conhecer mais sobre a arquitetura do que pode ser chamado de computação em nuvem.

Uma arquitetura de referência é encontrada em Bohn et al. (2011) e define cinco entidades (atores) :

- Consumidor da nuvem: um cliente (pessoa ou organização) que contrata os serviços de um provedor de nuvem.
- Provedor de nuvem: a entidade que disponibiliza o serviço de nuvem.
- Auditor da nuvem: uma equipe para avaliar os serviços oferecidos, operações, desempenho e segurança da implementação.
- Intermediador: uma entidade que gerencia os serviços e é responsável pelas negociações entre os provedores e os clientes.
- Suporte: garante a conectividade dos serviços entre provedores e clientes.

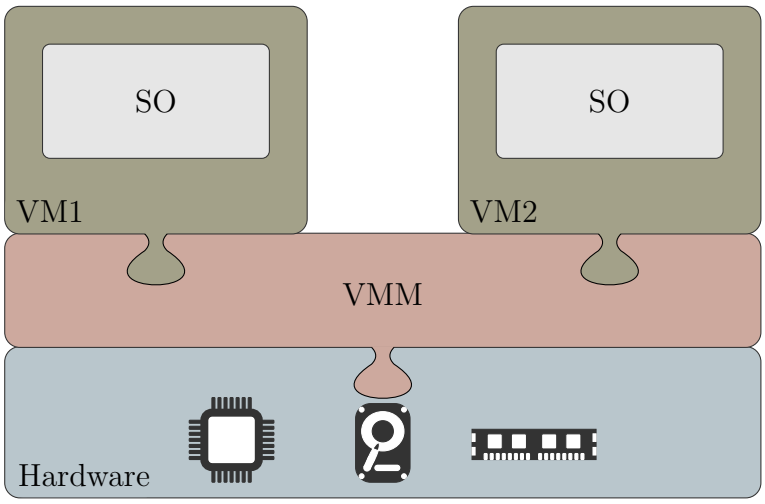
Além das entidades, um sistema de computação em nuvem é apresentado em duas seções diferentes - *front end* e *back end* - conectadas por uma rede, geralmente a internet (JADEJA; MODI, 2012). Isso significa que um cliente acessa o *front end* para realizar operações que serão executadas no *back end*, como acesso a um programa, um ambiente para programação ou uma VM.

3.1.1 Virtualização

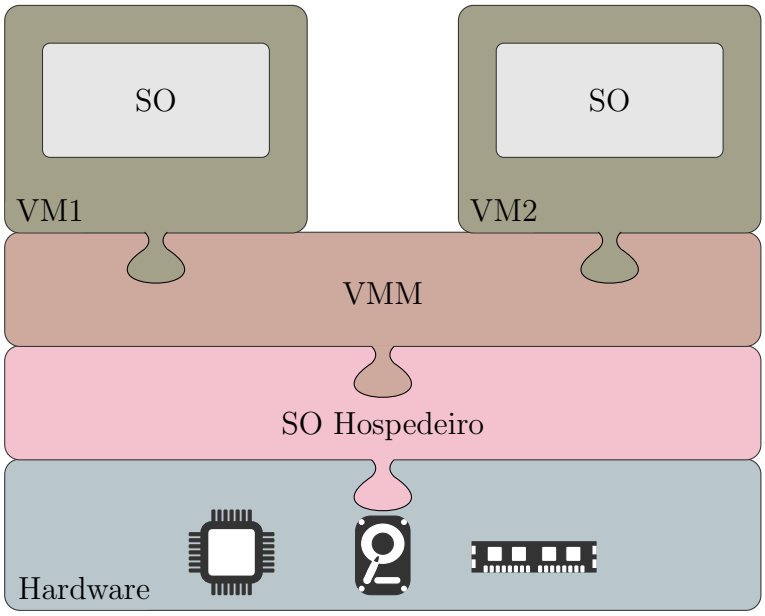
A virtualização é o componente principal da estrutura de uma nuvem. Para entender a importância, é preciso lembrar que quando um sistema é virtualizado, há um mapeamento dos recursos do sistema virtualizado para os recursos do sistema real (SMITH; NAIR, 2005). Dessa forma, é possível executar múltiplos sistemas virtuais dentro de um sistema, característica que permite o acesso de diversos usuários em uma única máquina física.

O hipervisor é o *software* responsável por mediar as interações das máquinas virtuais com o sistema hospedeiro. Para mediar, ele recebe requisições das VM, processa e envia para o *hardware* ou para o sistema operacional (SO), dependendo do tipo de VMM, obtém a resposta e a devolve à VM, criando uma abstração dos *drivers*. O VMM tem a vantagem de poder migrar máquinas virtuais entre máquinas, possibilitando o balanceamento de

carga entre as máquinas, além de ser capaz de suspender a atividade de uma VM e retomá-la mais adiante (ROSENBLUM; GARFINKEL, 2005). Sobre os tipos de VMM, há duas categorias principais - nativo e hospedado - que podem ser observadas na (Figura 4). O primeiro tipo (a) é executado diretamente no *hardware* em um modo privilegiado de execução; e o hospedado (b) executa sobre um sistema operacional (KIM, 2011).



(a) VMM nativo



(b) VMM hospedado

Figura 4: Tipos de VMM

Uma VM é o *software* que utiliza a abstração criada pelo VMM para suportar uma arquitetura desejada (SMITH; NAIR, 2005). Tais definições evidenciam diversas vantagens, sendo que uma das principais para um ambiente de computação em nuvem é a possibilidade de criar novas VMs em qualquer instante, com um monitor gerenciando essa criação, podendo enviá-las a qualquer computador em uma rede e a qualquer momento, possibi-

litando o balanceamento de carga, tolerância a falhas (quando uma VM de um usuário falha, os outros usuários não são prejudicados), e a utilização por múltiplos usuários, além de fornecer um ambiente de sistema operacional ou um item específico de *hardware* (como uma quantidade de armazenamento em disco rígido) de forma remota.

3.1.2 Serviços de nuvem

Considerando que os serviços em nuvem são virtualizados, diversas formas diferentes de se utilizar as VMs podem ser buscadas, ou seja, é possível ter-se apenas um *software* sendo executado em uma VM, uma biblioteca de classes e, como no caso desse trabalho, acesso completo ao sistema operacional e estrutura de rede. Tais serviços serão apresentados nas próximas seções.

3.1.2.1 Infraestrutura como serviço - IaaS

É o tipo de serviço que oferece o acesso completo à VM. Esse serviço é o foco desse trabalho e elimina a necessidade de comprar computadores servidores e equipamento de rede para executar um serviço *online* ou um programa paralelo ou distribuído. Entre as vantagens, está a possibilidade de pagar apenas pelo uso e a escalabilidade instantânea (JADEJA; MODI, 2012; MOLLAH; ISLAM; ISLAM, 2012), além de possibilitar a obtenção de imagens de VMs, facilitando a migração de dados entre nuvens (BOHN et al., 2011). O cliente desse serviço obtém acesso a processamento, armazenamento, compiladores, redes e qualquer outro serviço que um SO pode oferecer.

3.1.2.2 Plataforma como serviço - PaaS

Nesse serviço, o cliente pode desenvolver programas com bibliotecas fornecidas pelo provedor, ou comprar algum, e enviar ao provedor, utilizando também as capacidades de armazenamento e processamento do servidor, mas sem o controle oferecido pelo IaaS (ERL; PUTTINI; MAHMOOD, 2013). A vantagem desse serviço é novamente eliminar a necessidade de comprar um *hardware* dedicado para um programa, geralmente um serviço de internet. Exemplos desse serviço são o Google Apps e o Microsoft Azure (MOLLAH; ISLAM; ISLAM, 2012).

3.1.2.3 Software como serviço - SaaS

No IaaS, o cliente possui acesso completo ao SO, no PaaS o cliente tem acesso apenas às ferramentas de programação e no SaaS, o acesso é limitado apenas a *softwares* escolhidos pelo provedor. É importante observar que um serviço PaaS pode ser desenvolvido utilizando um provedor de IaaS e um serviço SaaS pode ser desenvolvido sobre um PaaS (CHHABRA; DIXIT, 2015).

3.1.3 Tipos de nuvem

Uma das principais questões que podem ser levantadas sobre os ambientes em nuvem é se existem formas diferentes de se instalar e configurar um ambiente. Outra questão que pode ser feita é se é possível interagir diversos ambientes de computação em nuvem. Para cada uma dessas abordagens, existe um tipo de conceito, como será visto mais adiante.

3.1.3.1 Nuvem Pública

Em uma nuvem pública, o serviço é oferecido ao público em geral em uma rede pública, e é mantido por uma organização que vende tais serviços (HUANG et al., 2015), semelhante ao conceito de compra de serviço de energia elétrica em uma casa (JADEJA; MODI, 2012).

Um exemplo de provedor de computação em nuvem é o Amazon EC2. Esse provedor oferece um serviço do tipo IaaS, ou seja, é muito semelhante a um *hardware* real (físico), permitindo ao usuário o controle de toda a pilha de *software* (ARMBRUST et al., 2009) e, juntamente com o Amazon S3, é possível realizar computações em *clusters* virtuais paralelos controlados sob demanda utilizando bibliotecas de paralelização em um sistema baseado em Linux (EVANGELINOS; HILL, 2008).

3.1.3.2 Nuvem privada

Embora a estrutura necessária para uma nuvem privada seja a mesma para uma nuvem pública, ela difere da primeira por não oferecer os serviços ao público em geral, mantendo o acesso apenas para os usuários de uma rede interna à empresa. Quanto à estrutura física, pode ser terceirizada ou mantida dentro da própria empresa (ROSADO; BERNARDINO, 2014). Exemplos de *softwares* utilizados para nuvens privadas são OpenStack (2015), CloudStack (2015) e VMWare vCloud (2015).

3.1.3.3 Nuvem comunitária

Esse tipo de nuvem é semelhante à nuvem privada com a diferença de ser um conjunto de organizações que compartilham infraestrutura e políticas de privacidade e segurança (LIU; VLASSOV; NAVARRO, 2014). Assim como a nuvem privada, uma nuvem comunitária pode compartilhar da nuvem de uma das organizações ou terceirizar o serviço (JADEJA; MODI, 2012).

3.1.3.4 Nuvem híbrida

Para se classificar como nuvem híbrida, dois tipos de nuvens distintas devem ser ligadas por alguma tecnologia padronizada ou proprietária (CARAGNANO et al., 2014). O uso de uma nuvem híbrida se faz necessário, por exemplo, quando é prevista a execução de uma tarefa muito custosa (JADEJA; MODI, 2012).

3.2 Desempenho

A configuração das máquinas virtuais nos provedores varia bastante com opções diferentes de quantidade de vCPU por VM, disco, transferência, RAM e desempenho de rede e, apesar de todas essas informações, o desempenho é instável, dependendo da arquitetura do servidor e da intensidade de uso de outros usuários na máquina física em que a VM do usuário está alocada, tornando as medidas apresentadas pelos provedores, insuficientes (LENK et al., 2011). Essa instabilidade pode ser observada com uma variação de dias (CLOUDSPECTATOR, 2012) ou entre pequenos intervalos de tempo (WANG; NG, 2010; ZOU et al., 2011).

Para o uso em computação científica, que é o objetivo desse trabalho, os serviços de nuvem mais populares não são uma opção vantajosa em relação ao desempenho, apesar de poder oferecer vantagens em relação ao preço (OSTERMANN et al., 2010; WALKER, 2008). O que isso quer dizer é que a computação de alto desempenho (HPC) vai levar mais tempo em um serviço de nuvem, seja ela baseada em troca de mensagens ou sequencial, mas o tempo de execução pode ser curto o suficiente para apresentar um custo menor do que a implementação de um *cluster*. Uma boa notícia é que os provedores, geralmente, oferecem opções de balanceamento de carga.

3.3 Instabilidade

Como mencionado anteriormente, o objetivo desse trabalho está intimamente ligado com a instabilidade dos ambientes de nuvem. Por causa da natureza dos serviços públicos de nuvem, existem diversos clientes utilizando a rede e máquinas físicas ao mesmo tempo e isso dificulta a garantia da estabilidade do serviço.

Existem diversos trabalhos que provam essa instabilidade, sendo que em Schad, Dittrich e Quiané-Ruiz (2010) é mostrado que o desempenho da CPU para a Amazon EC2 possui a relação entre o desvio padrão e a média, valor usado para comparar o grau de variação entre uma série de dados a outra - uma vez que o autor utilizou medidas de instâncias em máquinas localizadas em regiões geográficas distintas - coeficiente que é chamado de coeficiente de variação (CV) com valor de 24% para instâncias grandes (que são instâncias com valores altos de memória e processamento, definidos pelo provedor) e CV de 21% para instâncias pequenas, contra 0,1% em um *cluster* físico. Para a memória, foram obtidos os valores de CV de 10% em instâncias grandes e CV de 8% para instâncias pequenas, contra 0,3% em um *cluster* físico. Nos testes de leituras aleatórias e em blocos sequenciais de memória, o CV variou entre 9% e 20% contra 0,6% e 1,9% no *cluster* físico. Em largura de banda, os valores também foram altos, sendo 19% para as duas instâncias contra 0,2% do *cluster* físico. Além da discussão anterior, em CloudSpectator (2012) foi demonstrado que ocorrem variações dependendo do dia, semana ou hora em que o serviço é utilizado, apresentando, por exemplo, desvios de até 366.81 MIPS dentro de um ano. Mais do que isso, no trabalho de Dejun, Pierre e Chi (2009) é mostrado que o tempo de resposta em uma instância pequena varia em média de $185ms$ a $684ms$, sendo que os picos duram cerca de 2 minutos, fator atribuído pelos autores a uma provável alocação de máquina virtual em uma máquina física em comum. No artigo de Wang e Ng (2010) é demonstrada a diferença do tempo que uma mensagem leva para chegar ao destino somado ao tempo que a mensagem de confirmação levou para ser recebida, valor chamado de *round-trip time* (RTT), em 5000 medidas entre máquinas físicas e entre VMs em nuvem, com a nuvem apresentando muitas variações com valores entre $0,5ms$ e $20ms$ além de 10% das medidas apresentarem frequências de perda de pacote maiores que 10%.

Para se utilizar dos dados apresentados como um *delay* simulado, antes é preciso descrever uma equação para que os valores sejam devidamente calculados. Com base nos resultados apresentados por Schad, Dittrich e Quiané-Ruiz (2010), obtém-se σ = 640 × 0.19 = 121.6, o desvio padrão do desempenho de rede da Amazon EC2, a média μ = 640 e o coeficiente de variação CV = 0.19. A partir desses resultados e com a

informação, também do mesmo artigo, que esses dados obedeceram uma distribuição normal, chega-se à Equação 3.1. A equação para uma distribuição normal, apresentada na Equação 3.2 é descrita em Chwif e Medina (2014).

$$N(x) = \frac{1}{\sqrt{2\pi 121.6^2}} e^{-\frac{(x-640)^2}{2 \cdot 121.6^2}} \tag{3.1}$$

$$N(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.2}$$

A distribuição representada pela Equação 3.1 é ilustrada na Figura 5.

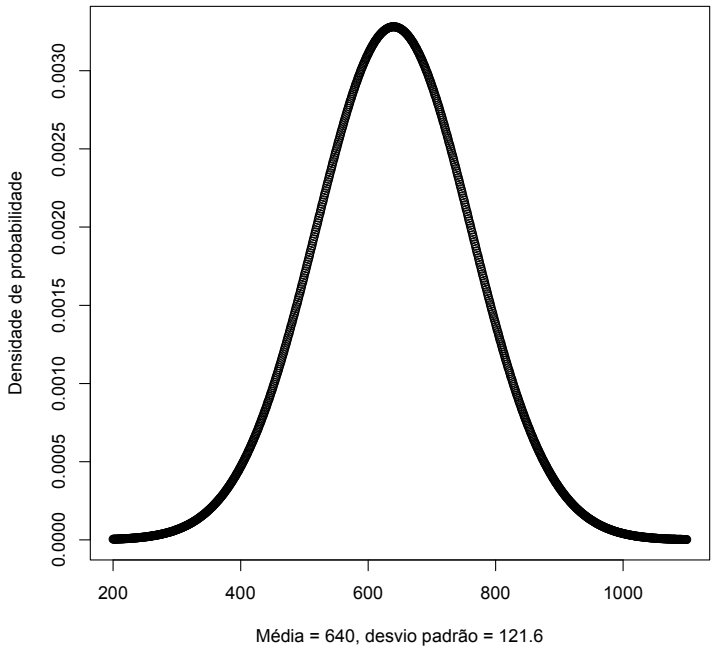


Figura 5: Distribuição normal obtida das medidas do desempenho na Amazon EC2

Com o uso da Equação 3.1, pode-se utilizar valores aleatórios para x e obter um valor de desempenho de rede para qualquer momento desejado. O gráfico da distribuição, para valores aleatórios de tempo, pode ser observada na Figura 6 e foi gerado para se aproximar dos dados obtidos e apresentados em Schad, Dittrich e Quiané-Ruiz (2010) (Figura 7).

3.4 Considerações finais

Embora o conceito de computação em nuvem esteja ligado ao mercado, principalmente pelo motivo de os serviços de computação em nuvem públicos serem pagos, seus princípios

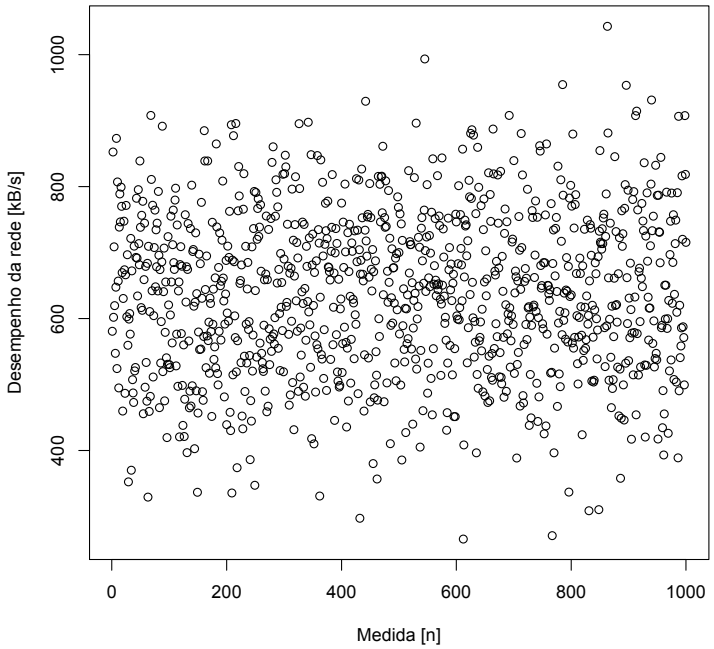


Figura 6: Distribuição do desempenho de rede

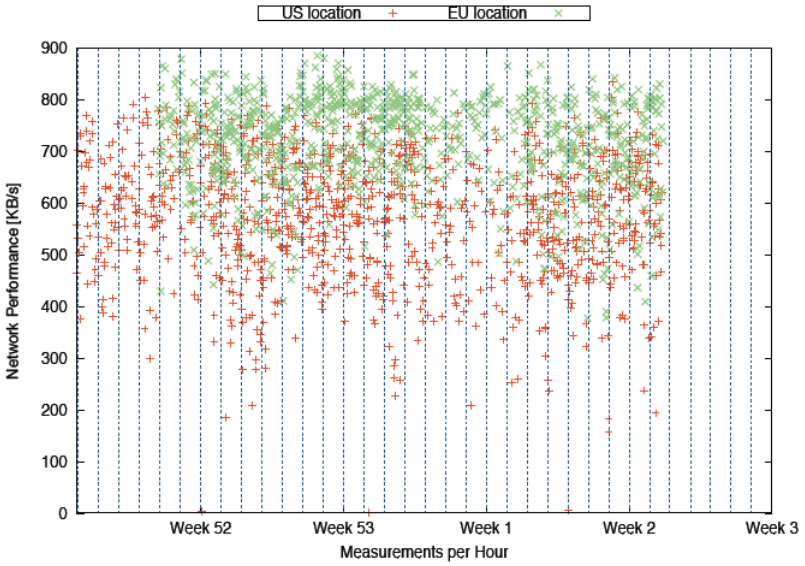


Figura 7: Desempenho de rede da Amazon EC2 (SCHAD; DITTRICH; QUIANÉ-RUIZ, 2010)

se aplicam a muitos casos obtidos na academia, pois permite um ambiente computacional completo de uma forma muito rápida, possibilitando inclusive a execução de simulações, distribuídas ou não. Um fator importante a se observar são os diferentes tipos de serviços, sendo que apenas os principais foram apresentados, pois esse fator é o que determina o tipo de ambiente que se deseja desenvolver.

Para se criar uma instância nos ambientes IaaS, geralmente, é apresentado ao cliente um método de seleção de configurações das VMs e o seu sistema operacional, possibilidade

que, como será visto adiante, influencia não só na velocidade, mas também na instabilidade dos recursos, informação que não é apresentada pelo provedor. Isso significa que, para uma simulação distribuída, seria viável escolher instâncias com configurações altas, ou seja, grandes quantidades de memória, número de processadores e configurações específicas de rede, se o provedor disponibilizar.

A instabilidade foi demonstrada baseada em medidas reais de trabalhos prévios e mostra que a rede de um serviço de nuvem com muitos usuários, como o *Amazon EC2*, pode causar um grande impacto no resultado final. A partir dos dados apresentados, foi proposta a equação 3.1, que será usada posteriormente para calcular atrasos em um simulador de ambientes em nuvem. Essa equação será usada nesse trabalho apenas como forma de prever a variação da instabilidade.

Considerando que todo o ambiente necessário para uma simulação distribuída pode ser criado nos serviços de computação em nuvem, resta analisar o impacto da instabilidade apresentada no resultado final da simulação, o que será discutido no próximo capítulo.

4 Simulação distribuída na computação em nuvem

Como visto anteriormente, os serviços do tipo IaaS oferecem paralelização de máquinas virtuais, acesso a rede, programação e bibliotecas de paralelização, sendo que isso é exatamente o que uma simulação distribuída necessita. Normalmente, utiliza-se *hardware* próprio, podendo-se aproveitar um processador de vários núcleos ou criar uma infraestrutura cara e complexa para esse fim. Utilizando serviços da nuvem, é possível economizar na criação e manutenção dessa infraestrutura, supondo que esses serviços sejam confiáveis e capazes de proteger dados confidenciais e proprietários (FUJIMOTO; MALIK; PARK, 2010).

Apesar da aparente fácil adaptação, é complicado analisar como seria o comportamento de uma simulação distribuída em nuvem, pois, como observado por D'Angelo e Marzolla (2014), ao analisar o algoritmo mais simples de sincronização, a sequência de passos da simulação só ocorre quando todos os processos lógicos da simulação completarem o passo anterior, tornando a velocidade de execução dependente do seu componente mais lento. Pensando dessa forma, em um ambiente cujo *hardware* é compartilhado com outros usuários, a simulação da interface com um sistema pode ser completa mas o *hardware* real – físico – pode não responder como se estivesse executando localmente.

Para executar uma simulação distribuída em nuvem, pode-se utilizar uma nuvem privada, caso o problema não seja a ausência de máquinas físicas, mas a necessidade de um acesso remoto ou o aproveitamento de algum recurso da nuvem, como a migração de VMs. Além da nuvem privada, pode-se utilizar uma nuvem pública, que é a opção que intensifica o problema da instabilidade e a posterior ocorrência de *"rollbacks"* por conta dos muitos atrasos com tempos muito variáveis. Como foi observado por Ledyayev e Richter (2014), o uso de computação de alto desempenho no *OpenStack*, que é utilizado normalmente para nuvens privadas, não possui um desempenho tão elevado quanto uma máquina física com a mesma capacidade, fator que deve ser levado em consideração, nos

casos em que o seu uso seria para justificar a mobilidade que o sistema permite. No caso de simulações distribuídas em nuvens públicas, como a *Amazon EC2*, que seria escolhido pela busca de um custo mais baixo, o desempenho pode ser alto, como demonstrado por Juve et al. (2009), mas a instabilidade causa um problema de degradação de desempenho, principalmente em instâncias com configurações mais baixas executando protocolos de sincronização, como mostrado em Vanmechelen, Munck e Broeckhove (2012), fatores que levaram à necessidade de se desenvolver novas abordagens, adequadas ao novo ambiente.

4.1 Adaptando a simulação distribuída à nuvem

A degradação de desempenho em protocolos de sincronização é inevitável, quando o ambiente é instável. No caso dos protocolos otimistas, é preciso se preocupar com a possibilidade de que uma mensagem enviada corretamente possa não ter atingido o seu destino, ou de alguma máquina estar com o desempenho prejudicado, como analisado em Malik, Park e Fujimoto (2010), em que um novo protocolo, baseado no *Time Warp*, foi desenvolvido para ajustar dinamicamente a execução de cada processo lógico, baseado em parâmetros locais. A principal alteração, em relação ao *Time Warp*, é mandar, periodicamente, mensagens chamadas de *termed heartbeat* - que contém informações sobre mensagens enviadas - aos processos lógicos para receber informações desses processos, residentes em outras máquinas. Assim, a mensagem desgarrada pode ser identificada, evitando *rollbacks* frequentes devido a cargas assimétricas de processamento.

Outro protocolo, apresentado em Fujimoto, Malik e Park (2010), é o Aurora. Embora ele não tenha sido desenvolvido para esse fim, seu uso é justificado devido ao seu funcionamento. Esse protocolo utiliza o paradigma de mestre/trabalhador e divide computações potencialmente grandes em porções menores de trabalho e distribui essas porções em um *pool* de processos trabalhadores que executam de acordo com o controle de um mestre, que já é utilizado em computação distribuída (Figura 8). Com esse paradigma, ao invés de trocar muitas pequenas mensagens entre processos lógicos, o mestre transmite mensagens para os trabalhadores e aguarda os resultados de todos. Nesse protocolo, o mestre é dividido em três serviços principais: *Aurora Proxy*, controlador central que supervisiona a simulação e os dois outros serviços, servidores de estado de unidade de múltiplos trabalhos e servidores de estado de mensagens de unidade de múltiplos trabalhos. Como análise, apenas foi observado o impacto da granularidade da unidade de trabalho em relação à quantidade de trabalho por locação de unidade de trabalho, no rendimento global do sistema, e o resultado foi favorável à granularidade grossa, já que a granularidade fina

transmite mensagens constantemente.

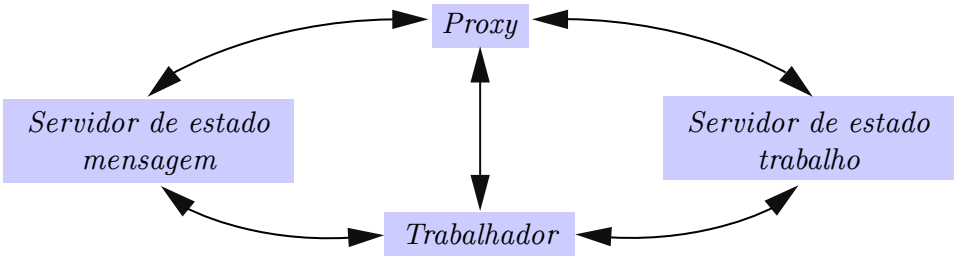


Figura 8: Protocolo Aurora

Além dos trabalhos demonstrados no artigo de Fujimoto, Malik e Park (2010) e Malik, Park e Fujimoto (2010), existem tentativas de adaptação de simulações distribuídas baseadas em *middleware* de simulação, que é um modelo muito utilizado, mas também possui problemas de adaptação à nuvem, por conta da forma ineficiente de utilização de recursos, falta de balanceamento de carga e fraca tolerância a falhas, como observado por He et al. (2012), onde foi desenvolvida uma extensão da *High Level Architecture* (*HLA*), arquitetura que possibilita a troca de dados entre objetos de uma simulação, através de serviços da *Run-Time Infrastructure* (*RTI*) (IEEE STANDARD, 2010). Nessa extensão, novas funcionalidades foram implementadas no componente Cloud-RTI. Essa implementação adota a invocação baseada em agentes e a estratégia *callback*. O funcionamento do sistema consiste de quatro partes: Invocação da interface de visualização HLA – que provê interfaces de serviços de nuvem –, agente de requisição de processo, agente de *callback* Cloud-RTI e Cloud-RTI. Resumidamente, quando ocorre uma requisição do processo, a mensagem é entregue ao Cloud-RTI e o agente de *callback* recebe o *callback* do Cloud-RTI, gerando os eventos e os colocando na fila.

Uma outra tentativa de adaptação às nuvens é apresentada em D'Angelo e Marzolla (2014), em que foram desenvolvidos *middleware* e *framework* para simulação distribuída. O *middleware* cujo nome é Advanced RTI System (ARTIS), provê serviços de tempo de execução. O *framework*, chamado de GAIA+, disponibiliza serviços tais como as primitivas de comunicação para as entidades de simulação. Além disso, foi adicionada uma nova camada, chamada de *GAIA Fault-Tolerance* que permite execuções tolerantes a falhas, mas que até então não foi avaliado e testado completamente.

Os trabalhos discutidos apresentam melhorias para casos em que se deseja executar uma simulação em uma nuvem pública, pois mencionam problemas e propõem alterações baseados em casos estudados em um ambiente público, como o *Amazon EC2*. Os ambientes em nuvem possuem, apesar das desvantagens, diversas vantagens. Dependendo

das vantagens que se deseja explorar, pode ser interessante utilizar as funcionalidades dos ambientes em nuvem, mas em um sistema computacional próprio, caso que será discutido brevemente a seguir.

4.2 Simulação em nuvem privada

Nos casos em que o interesse no uso de computação em nuvem esteja relacionado com a disponibilidade aos clientes, segurança, confiança e usabilidade, uma boa solução é a criação de uma nuvem privada, como demonstrado por Guan (2015). A primeira grande vantagem da nuvem privada é a certeza de quais recursos estão sendo utilizados em um dado momento, eliminando o grande problema da instabilidade imprevisível das nuvens públicas. A segunda vantagem é a possibilidade de migrações de VMs, pois é possível balancear o uso dos recursos migrando não os processos mas a VM, caso se queira organizar as simulações de vários clientes. Outras vantagens que podem ser citadas são a escalabilidade e a disponibilidade, pois uma atualização de *hardware* seria completamente transparente aos usuários e novas instâncias poderiam ser criadas imediatamente, agilizando a implementação de uma nova simulação.

Para criar uma nuvem privada, é preciso instalar um *software* de gerência sendo que versões de código aberto, como OpenStack (2015) e CloudStack (2015) oferecem suporte a *hypervisors* populares como KVM (2015), VMWare (2015) e XenServer (2015). O projeto OpenStack oferece uma versão que pode ser instalada em apenas uma máquina, para analisar os recursos, ao contrário do CloudStack que exige uma estrutura mais elaborada. Ambos oferecem duas possibilidades - linha de comando ou interface de navegador - para gerenciar as máquinas virtuais, discos virtuais e rede. Independente da ferramenta, após instalada, o ambiente virtual de rede, máquinas e discos, oferece acesso total a qualquer recurso do sistema operacional, possibilitando assim a execução da simulação ou aplicação.

Com a instalação de uma nuvem privada, o acesso é a uma nuvem IaaS, mas para transformar em uma nuvem PaaS ou SaaS depende do programador, que pode criar uma biblioteca para a criação de novas simulações e liberar o acesso apenas à API ou uma interface de navegador para a criação visual de novas simulações. A maior desvantagem nessa abordagem seria abrir o sistema para muitos clientes, gerando o mesmo problema de instabilidade das nuvens públicas, a não ser que fosse um serviço oferecido com um controle maior de acesso. Além disso, o próprio sistema pode ser responsável por balancear os processos das simulações entre as máquinas, utilizando os comandos oferecidos pelos

softwares para o gerenciamento das VMs.

4.3 Considerações finais

Embora as tentativas de simulação em nuvem apresentadas até a presente data não sejam tão numerosas, é possível observar que o tema está atraindo a atenção de diversos grupos, como discutido em Jafer, Liu e Wainer (2013), onde é concluído que ainda é necessário pesquisar mais para se conseguir simulações distribuídas de alto desempenho em nuvem.

Para cada tipo de nuvem existe uma vantagem de se executar simulações no ambiente, pois somente em uma nuvem pública é possível fazê-lo para a diminuição de custos, enquanto que em uma nuvem privada ainda existe a vantagem do acesso remoto, mas o desempenho fica abaixo da execução diretamente na máquina física. Apesar das vantagens, o fator que preocupa é em relação ao tempo, pois executar uma simulação distribuída em nuvem pode se tornar menos vantajoso, uma vez que o tempo de execução pode ser muito elevado, principalmente se forem utilizadas instâncias com configurações baixas.

Mesmo apresentando problemas a serem resolvidos, é possível encontrar projetos que visam criar serviços de simulação em nuvem, ou seja, projetos que não visam o estudo do caso ou executar apenas uma simulação própria, mas disponibilizar um ambiente de simulação como serviço ao usuário, como o *SimPaaS*, encontrado em Simzentrum (2015), que visa o desenvolvimento de um serviço PaaS que permita que simulações sejam implementadas como SaaS. Esse projeto escolheu o *sofware OpenStack* para o gerenciamento de nuvem, não sendo, portanto, desenvolvido em cima de uma nuvem pública e propõe, como medida para melhoria de eficiência, melhorias no escalonamento de VMs e migração de VMs durante a simulação, além de diminuição das latências da rede e virtualização e aumento da prioridade dos serviços de nuvem no sistema operacional. Outro projeto é apresentado por Mancini et al. (2012), que busca um sistema em que o servidor em nuvem execute a simulação, enquanto um aparelho móvel com *Android* instalado, permita visualizar os dados e gerenciar a simulação.

O que se pode observar nos trabalhos estudados é que todos possuem problemas em relação aos testes, que devem ser feitos em diversos tamanhos de instância, diversas datas e horários diferentes e diferentes serviços de computação em nuvem, dificultando muito o desenvolvimento. Por causa desse problema, esse trabalho propõe o desenvolvimento de uma ferramenta de simulação de nuvem, para algoritmos de simulação distribuída,

que permita que os fatores de atraso sejam modificados e controlados e que possua um controle maior dos caminhos das mensagens, possibilitando que análises sejam feitas mais facilmente. Essa proposta será apresentada no próximo capítulo.

5 NetworkCloudSim para análises em Simulação Distribuída

Para analisar os problemas relacionados aos sucessivos *rollbacks*, é necessário um ambiente controlado, não necessariamente estável mas com uma instabilidade controlada, baseada em dados apresentados por outros trabalhos. Apesar de parecer impreciso, a própria natureza da nuvem é imprecisa, já que seu desempenho varia em curtos e longos intervalos de tempo, ou seja, uma mensagem ou processamento pode ter diferenças de tempo entre medidas diferentes e pode levar um tempo médio muito maior dependendo do dia ou da hora em que for executado.

Para prever e desenvolver algoritmos que diminuam o número de *rollbacks*, é interessante possuir um *software* em que a instabilidade possa ser intensificada pelo programador, para induzir um comportamento favorável ao *rollback* e, assim, trabalhar em uma técnica para combatê-lo. Para isso, alterou-se o Network CloudSim (CLOUDSIM, 2015), implementando-se classes e funções, para aproveitar a estrutura de redes e máquinas virtuais disponíveis na ferramenta, para transmitir pacotes e executar processos, a partir das novas classes adicionadas, como será detalhado nas próximas seções.

Desenvolver sobre o NetworkCloudSim não é uma decisão direta, já que deve ser considerada a possibilidade de testar uma simulação sobre um ambiente real de nuvem. Para isso, primeiramente, foi testado o CloudStack e, logo após, o OpenStack, que são *softwares* para a criação de nuvens privadas. Apesar de ambos serem *softwares* muito bons e funcionais e que possibilitam a utilização de mais de um tipo de VMM (não simultaneamente), o problema surge na hora de simular instabilidades na rede ou no processamento, já que em uma nuvem privada, não existem usos de muitos usuários diferentes, e mesmo que fossem criados usuários diferentes, não seria possível fazer a equivalência do comportamento de uma nuvem comercial. Descartada a hipótese de se utilizar uma nuvem privada, foi considerada a possibilidade da "simulação da simulação" em um simulador de nuvem, e que foi usada ao final. A maior vantagem, como já mencionado, é o controle das situações

para a formulação de hipóteses, além de tornar mais acessível o *software* utilizado para a geração dos resultados, porque se fosse uma simulação ou um conjunto de simulações, necessitaria que outro usuário também testasse em um ambiente de nuvem.

5.1 NetworkCloudSim

O CloudSim é um *framework* de simulação que permite modelar, simular e experimentar infraestruturas de computação em nuvem (BUYYA; RANJAN; CALHEIROS, 2009). Com ele, é possível especificar configurações para as máquinas virtuais, definir a estrutura de rede, alocar diversas máquinas virtuais e analisar o tempo de escalonamento e criação das máquinas, entre outros recursos como a disponibilidade do núcleo de virtualização.

A partir do CloudSim, foi desenvolvido o NetworkCloudSim, que é um *framework* que tem por objetivo simular centros de dados de ambientes de nuvem e aplicações em máquinas virtuais na nuvem. Com ele, é possível modelar um *datacenter*, incluindo os elementos de processamento, memória, *switchs* e máquinas virtuais. Como este *framework* foi a base para o desenvolvimento desse trabalho, a seguir serão analisadas as camadas dessa ferramenta, com foco no funcionamento das máquinas virtuais, tarefas dadas às VMs (*cloudlet*) e envio das mensagens.

5.1.1 Arquitetura do NetworkCloudSim

O NetworkCloudSim é um ramo do projeto CloudSim e possui a arquitetura mostrada na Figura 9. Pode-se observar que a arquitetura possui diversas camadas (Código do usuário, *CloudSim/NetworkCloudSim* e o núcleo de simulação), que serão discutidas a seguir:

- Código do usuário: responsável pela definição das configurações da simulação.
- NetworkCloudSim: possui diversas camadas internas, que possuem a modelagem do serviço de nuvem, como a topologia de rede, recursos de nuvem (centro de dados, coordenador e tratamento de eventos), serviços da nuvem (alocação de memória, CPU, armazenamento e banda), serviços da máquina virtual (gerenciamento e execução) e estruturas de interface (máquina virtual, aplicação e tarefas).
- Núcleo de simulação: responsável pela interação entre as entidades (responsáveis por lidar com eventos) e os componentes (que se comunicam por envio de mensagens).

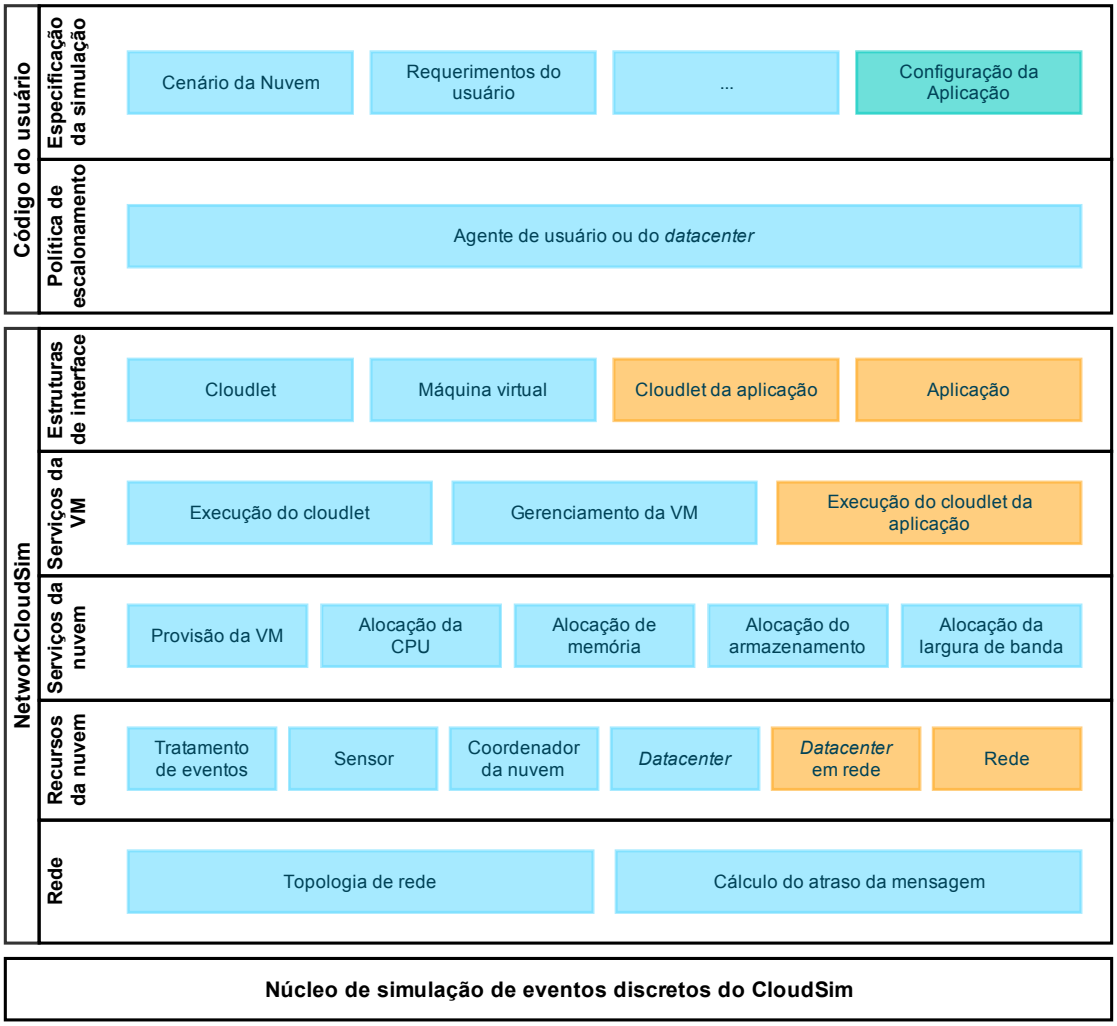


Figura 9: Arquitetura do NetworkCloudSim (GARG; BUYYA, 2011)

Para entender a criação de uma nova simulação nessa ferramenta, resumidamente, o usuário deve descrever as especificações do ambiente em nuvem, criar um agente para alocar os recursos e descrever o fluxo de trabalho de cada *cloudlet* que será usado. Com todos os recursos descritos e criados, a simulação inicia e os estágios criados na descrição do fluxo de trabalho são executados apropriadamente, permitindo o envio e recebimento de mensagens entre máquinas virtuais. Cada mensagem enviada passa pela rede do simulador (o objeto lógico e não o *hardware*) e é entregue à *cloudlet* a qual foi destinada, que remove da fila e pode enviar uma resposta, aguardar um tempo ou receber uma nova mensagem. O atraso das mensagens é calculado pelo simulador e a mensagem só é entregue após o encerramento do período calculado.

5.1.2 Descrição dos recursos

A descrição dos recursos deve ser feita para a criação do *hardware* e das máquinas virtuais. As classes utilizadas são as apresentadas na Figura 10, onde se observa que a classe principal (**UserApp**), criada pelo usuário, é responsável por instanciar todos os elementos da nuvem, que são passados ao *broker*, responsável por criar as máquinas virtuais a partir desses recursos.

A demonstração completa da descrição dos recursos pode ser analisada no Apêndice A, mas, resumidamente, são criados os objetos de elemento de processamento (*Pe*), os objetos de *NetworkHost*, que representam a máquina hospedeira e possuem as listas com os *Pe*, o centro de dados (*NetworkDatacenter*), que contém o objeto que representa suas características (*DatacenterCharacteristics*), além de uma lista de *NetworkHost*. Um objeto do tipo *DatacenterCharacteristics* será utilizado pelo *Datacenter*. Após o centro de dados ser criado, é necessária a criação da estrutura de rede, que pode ser realizada com a criação de *switches*, com a utilização das classes do tipo *RootSwitch*, *EdgeSwitch* ou *AggregateSwitch*. Para conectar os *NetworkHost* e os *switches*, deve-se criar um *broker*, que está representado na Figura 10 como *MyDatacenterBroker*, que incluirá também uma lista de máquinas virtuais (*NetworkVm*). A criação e descrição das características das máquinas virtuais são feitas no método *CreateVMs*, dentro da classe *MyDatacenterBroker*.

5.1.3 Criação das tarefas (Cloudlets)

Anteriormente, foi descrita e criada a estrutura do *datacenter*. Para executar um *workflow*, que é a estrutura que simula o envio e recebimento de mensagens entre VMs, é preciso descrever as tarefas (**Cloudlets**) e os estágios das tarefas (**TaskStage**), que enviarão mensagens entre si. O diagrama de classes dessa etapa é mostrado na Figura 11.

Nesse ponto da utilização do *framework*, é criada uma classe, estendida de *AppCloudlet* (chamada de *MyAppCloudlet*, na figura 11), para a criação dos estágios (*TaskStage*) e tarefas (*NetworkCloudlet*), que deve ser instanciada no método *createVmsInDatacenterBase*, da classe *MyDatacenterBroker*. A demonstração completa da criação dos *Cloudlets* é observada no Apêndice A.

A partir dessa breve análise da criação de uma simulação nesse *framework*, foi evidenciado que o *NetworkCloudSim* é uma boa opção para analisar e otimizar o desempenho de uma infraestrutura de nuvem, pois possui a modelagem que será necessária para esse trabalho. Apesar de o recurso de *workflow* ajudar a analisar a troca de mensagens entre

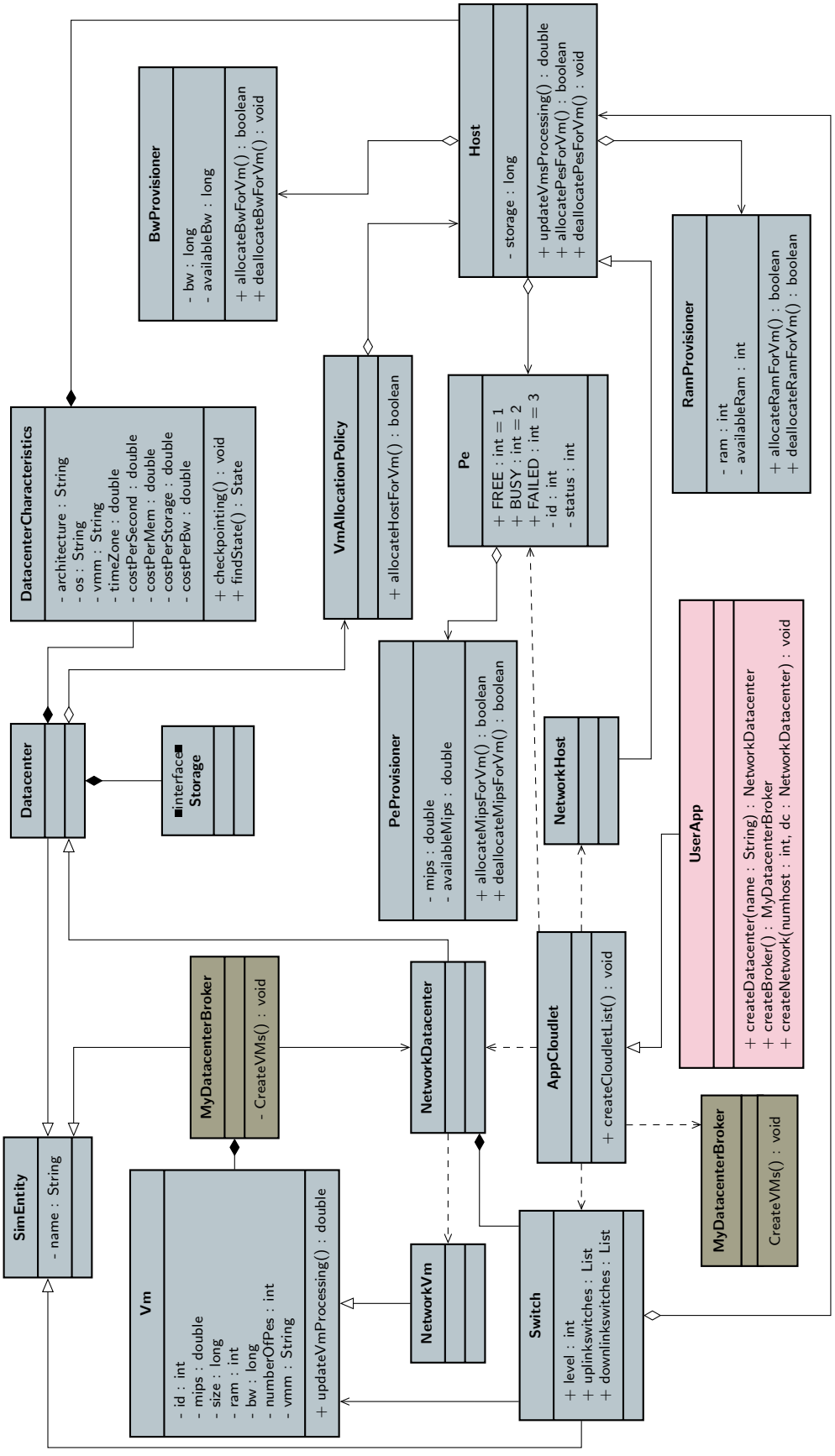


Figura 10: Diagrama parcial do NetworkCloudSim - Criação de recursos

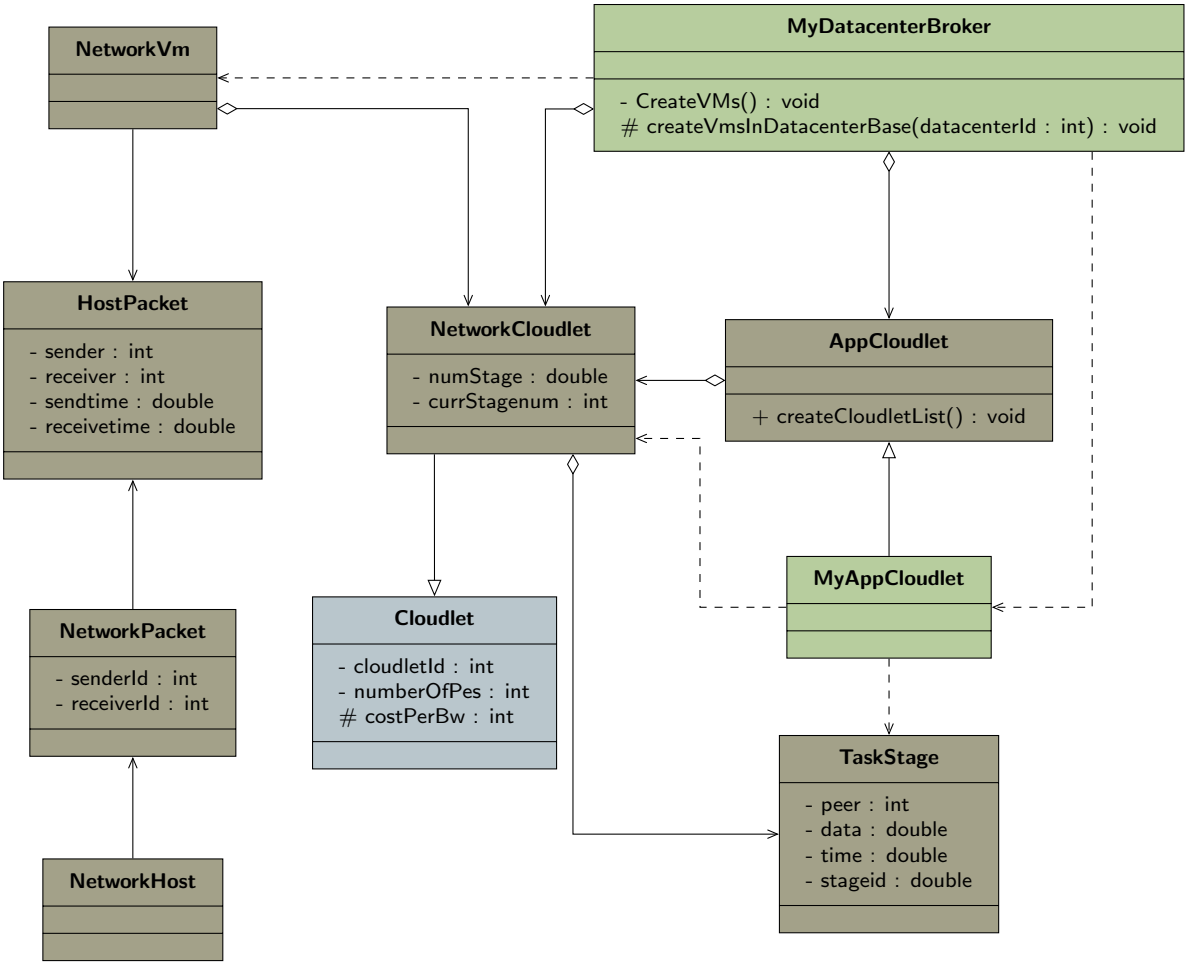


Figura 11: Diagrama parcial do NetworkCloudSim - Criação de cloudlets

máquinas virtuais, ele não foi modelado para simular o funcionamento de uma rede real. O problema é que as mensagens do *workflow* são enviadas e recebidas de uma forma sequencial, forçando que cada tarefa seja executada até o final e que as tarefas sejam executadas na mesma ordem. Além disso, o *NetworkCloudSim* não considera o principal problema enfrentado pelas simulações distribuídas, que é a instabilidade do desempenho. Levando esses pontos em consideração, foram feitas diversas alterações no projeto original, para que fosse possível simular um protocolo de simulação distribuída. Tais alterações serão discutidas a seguir.

5.2 NetworkCloudSim para Simulação Distribuída

A proposta do trabalho é disponibilizar uma ferramenta que permita a análise dos algoritmos dos protocolos de simulação distribuída em um ambiente controlado, podendo observar todo o trajeto das mensagens e o tempo de atraso de cada uma e então poder

prever alterações no algoritmo para aumentar a eficiência em uma simulação em nuvem. A proposta de um *framework* para simulação distribuída é apresentada em Cruz (2009). Para desenvolver a ferramenta desse trabalho, os diagramas apresentados no trabalho de Cruz (2009) (Figuras 12 e 12) serão adaptados ao *NetworkCloudSim* e os atrasos serão alterados de acordo com a equação 3.1. Para que isso seja possível, será necessário fazer com que os objetos que representam os processos, sejam executados como *threads*. Além disso, o modelo de *workflow* apresentado pelo *NetworkCloudSim* não se fez necessário, pois os envios e recebimentos são feitos imediatamente, sem uma programação prévia dos momentos em que cada um deve ocorrer.

5.2.1 Framework para Simulação Distribuída

Como mencionado, o trabalho de Cruz (2009) apresenta um *framework* para simulação distribuída. Adaptar completamente o *framework*, apresentado na Figura 12, é inviável, já que foi projetado para casos reais de simulação e não a mesma executando sobre um simulador. A adaptação também seria redundante, pois o *NetworkCloudSim* disponibiliza classes para distribuições de números aleatórios (**ContinuousDistribution**), arquitetura (**DatacenterCharacteristics**), mensagem (**HostPacket**) e para o ambiente (**NetworkDatacenter**).

Além do *framework*, ainda é necessário a adaptação dos protocolos. A proposta do *framework* para o protocolo *TimeWarp* é demonstrada na Figura 13 e será menos alterada, embora o observador não tenha sido utilizado, por se tratar da implementação do *TimeWarp*, protocolo que não necessita de tal abstração. Além disso, a mensagem será substituída pelo **HostPacket**. A seguir, o *framework* será descrito brevemente, para compreender as alterações realizadas nesse trabalho.

- **Environment**: contém os objetos principais da aplicação e serve para diferenciar o ambiente físico, ambiente de troca de mensagens, protocolo de simulação e o modelo.
- **GeneralState**: contém informações sobre o comportamento da simulação.
- **Architecture**: responsável pela descrição lógica da arquitetura física.
- **Model**: classe abstrata da qual os modelos, como redes de filas e redes de petri, devem ser estendidos na implementação.
- **RandomNumbers**: é a classe em que os diferentes algoritmos de distribuição de probabilidade devem ser implementados.

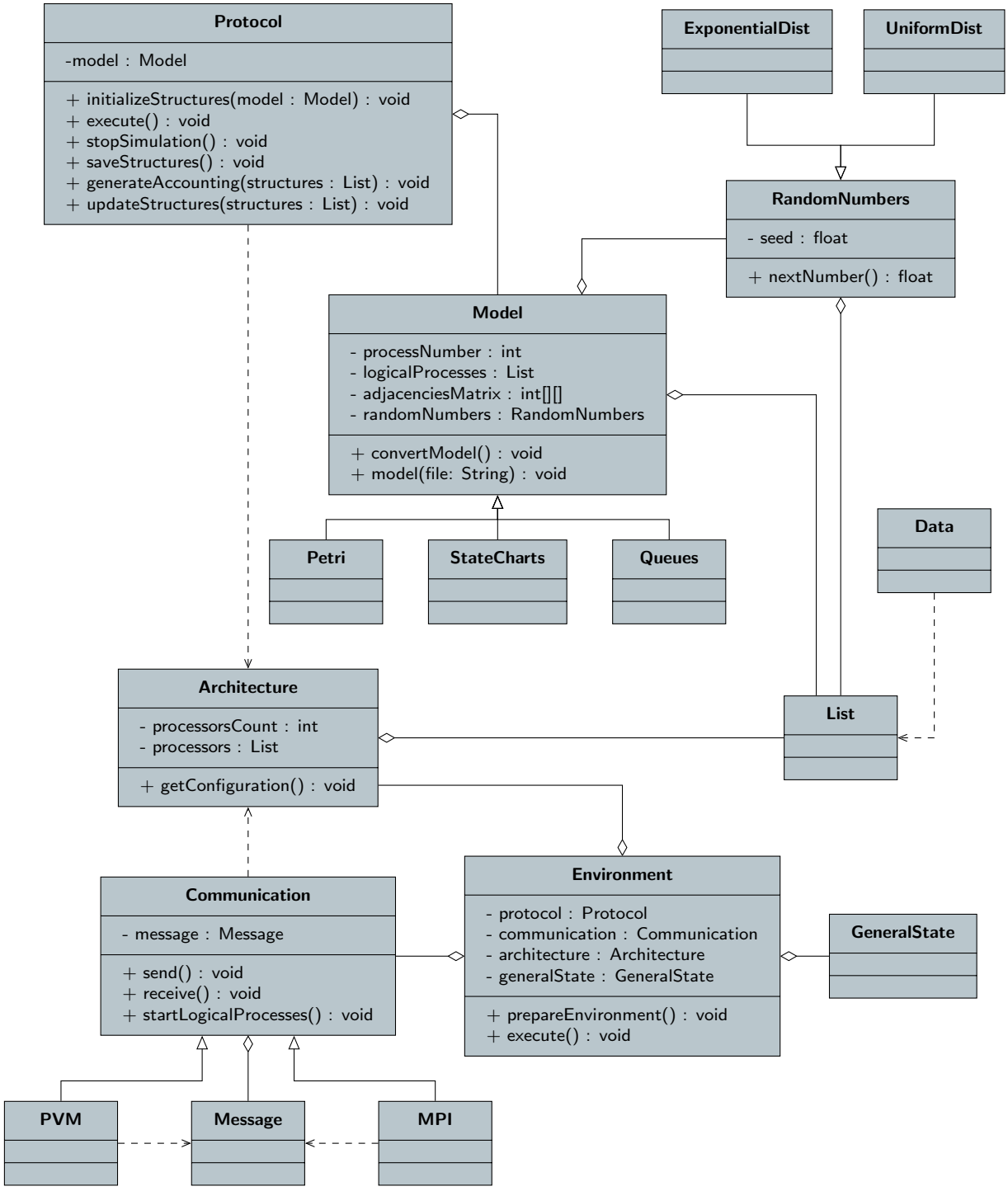


Figura 12: Diagrama do framework (CRUZ, 2009)

- **Communication**: classe que deve ser estendida para se implementar os diferentes ambientes de troca de mensagens, para tornar a escolha do ambiente transparente ao protocolo.

- **Message**: representa os dados que serão trocados pela rede na simulação e deve ser estendida para o uso desejado.

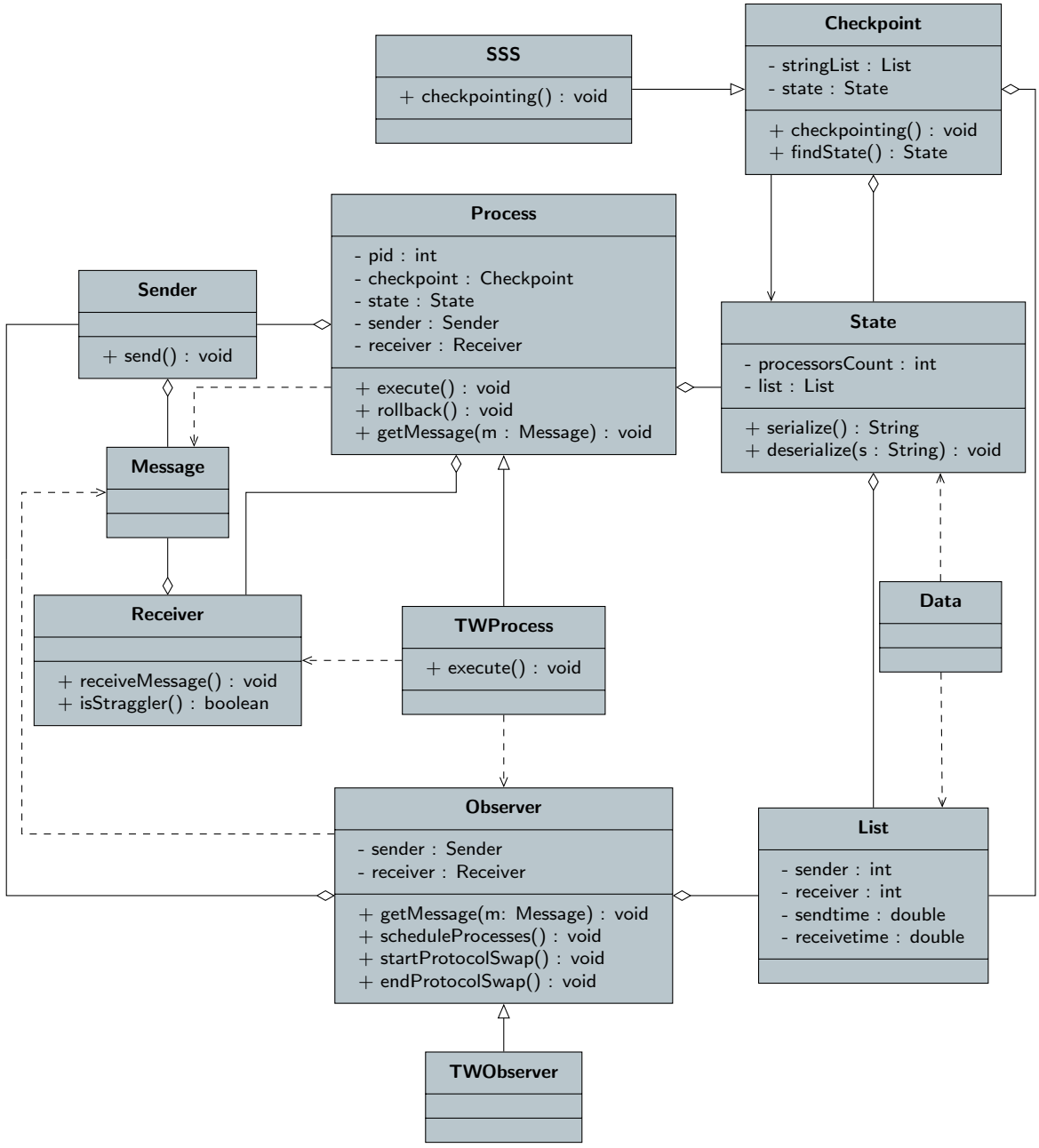


Figura 13: Diagrama do protocolo TimeWarp (CRUZ, 2009)

- **Protocol**: é a classe que faz a conexão entre o *framework* e o protocolo implementado.

Além do *framework*, foi apresentado um diagrama possível para o protocolo *TimeWarp* (Figura 13) e algumas classes serão descritas brevemente a seguir.

- **Process**: é uma classe abstrata, da qual dois tipos podem estender - processo com observador e sem observador - sendo que no diagrama apresentado foi demonstrado apenas o caso com observador e nesse trabalho foi escolhida uma abordagem sem

observador;

- **Checkpoint**: também é uma classe abstrata, da qual devem estender os diferentes tipos de algoritmos para *checkpoint*.
- **State**: os estados salvos durante um *checkpoint* são representados por essa classe, que contém atributos dos processos em questão.

Dada a explicação do *framework*, e considerando a apresentação do *NetworkCloudSim* fornecida anteriormente, é possível compreender o diagrama proposto na Figura 14, no qual diversas alterações foram feitas para melhor se adaptar ao simulador.

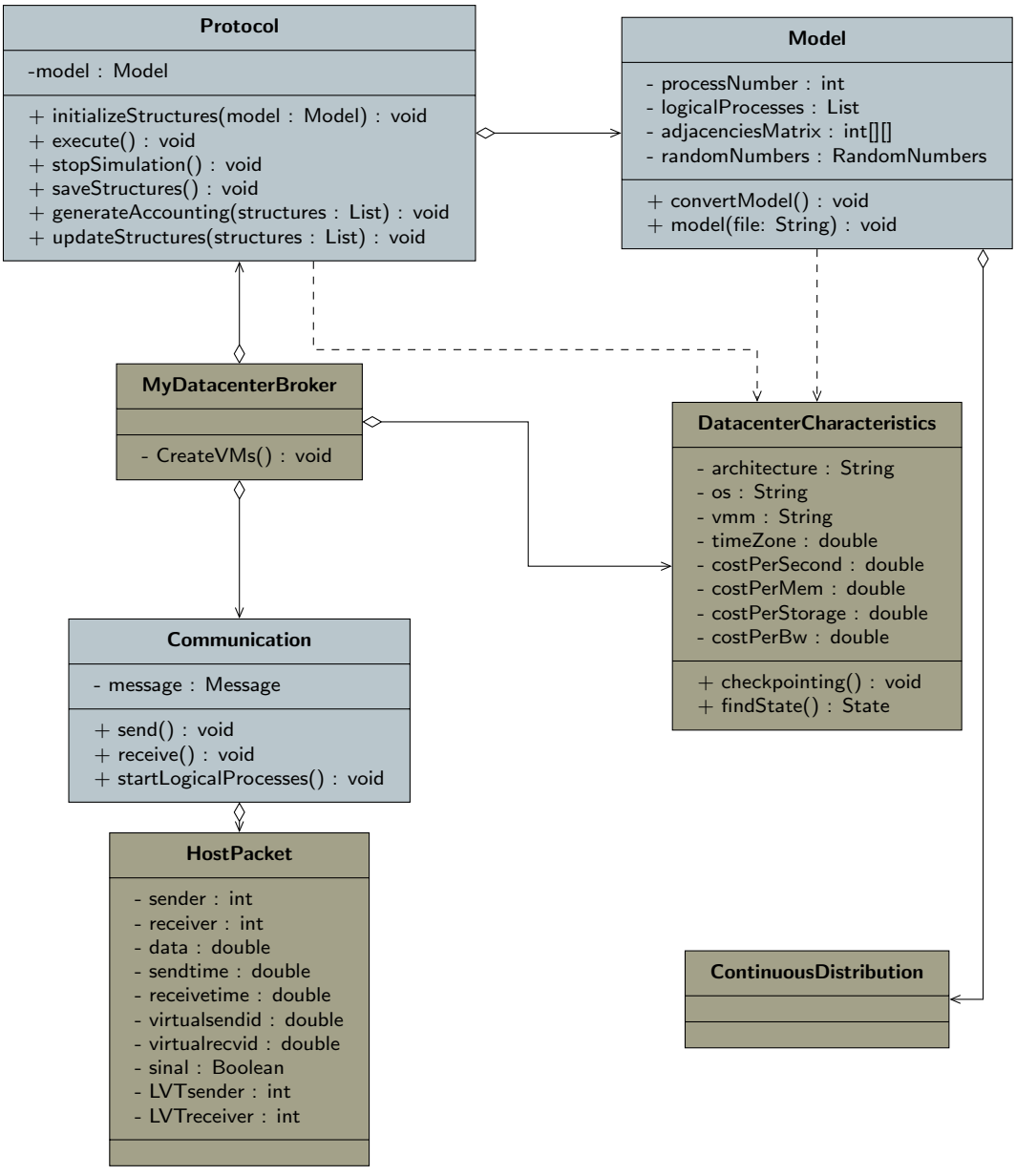


Figura 14: Diagrama do framework adaptado ao NetworkCloudSim

A substituição mais direta foi a troca da implementação da classe **Mensagem** pelo uso da classe **HostPacket**, que possui os campos:

- **sender**: identificador da VM que hospeda o processo que enviou a mensagem;
- **reciever**: identificador da VM que receberá a mensagem;
- **data**: dado que será enviado com a mensagem, se houver;
- **sendtime**: tempo, no relógio do *NetworkCloudSim*, que ocorreu o envio;
- **recievetime**: tempo de recebimento;
- **virtualsendid**: identificador do *cloudlet* que enviou a mensagem;
- **virtualrecvid**: identificador do *cloudlet* que receberá a mensagem.

Além dos seguintes campos adicionais, acrescentados nesse trabalho:

- **sinal**: sinal da mensagem, de acordo com o protocolo *TimeWarp*;
- **LVTsender**: tempo virtual local do remetente;
- **LVTreceiver**: tempo virtual local do receptor.

As distribuições de números aleatórios também já estavam presentes no *NetworkCloudSim* e nenhuma alteração foi necessária. A classe **DatacenterCharacteristics** já contém informações de arquitetura e substituiu a classe **Arquitetura**. Por último, a classe **Ambiente** foi substituída pela **MyDatacenterBroker**, pois essa também contém os diferentes objetos do ambiente. A comunicação é feita pela manipulação dos *maps* disponibilizados - *pkttosend* para os pacotes a enviar e *pktrecv* para os pacotes recebidos - pelo escalonador. Esses pacotes são enviados posteriormente pela classe **NetworkHost** (apresentada no Apêndice A) no método **sendpackets** e recebidos pelo método **recvpackets** da mesma classe.

Para a implementação do protocolo, foi utilizada a modelagem da Figura 15, em que os objetos **TWProcess** são *threads* executadas junto com o início do simulador e é nessa classe que o algoritmo principal do protocolo é implementado. Para tornar o algoritmo funcional, diversas alterações foram feitas no código original do *NetworkCloudSim*, sendo que as principais serão descritas adiante.

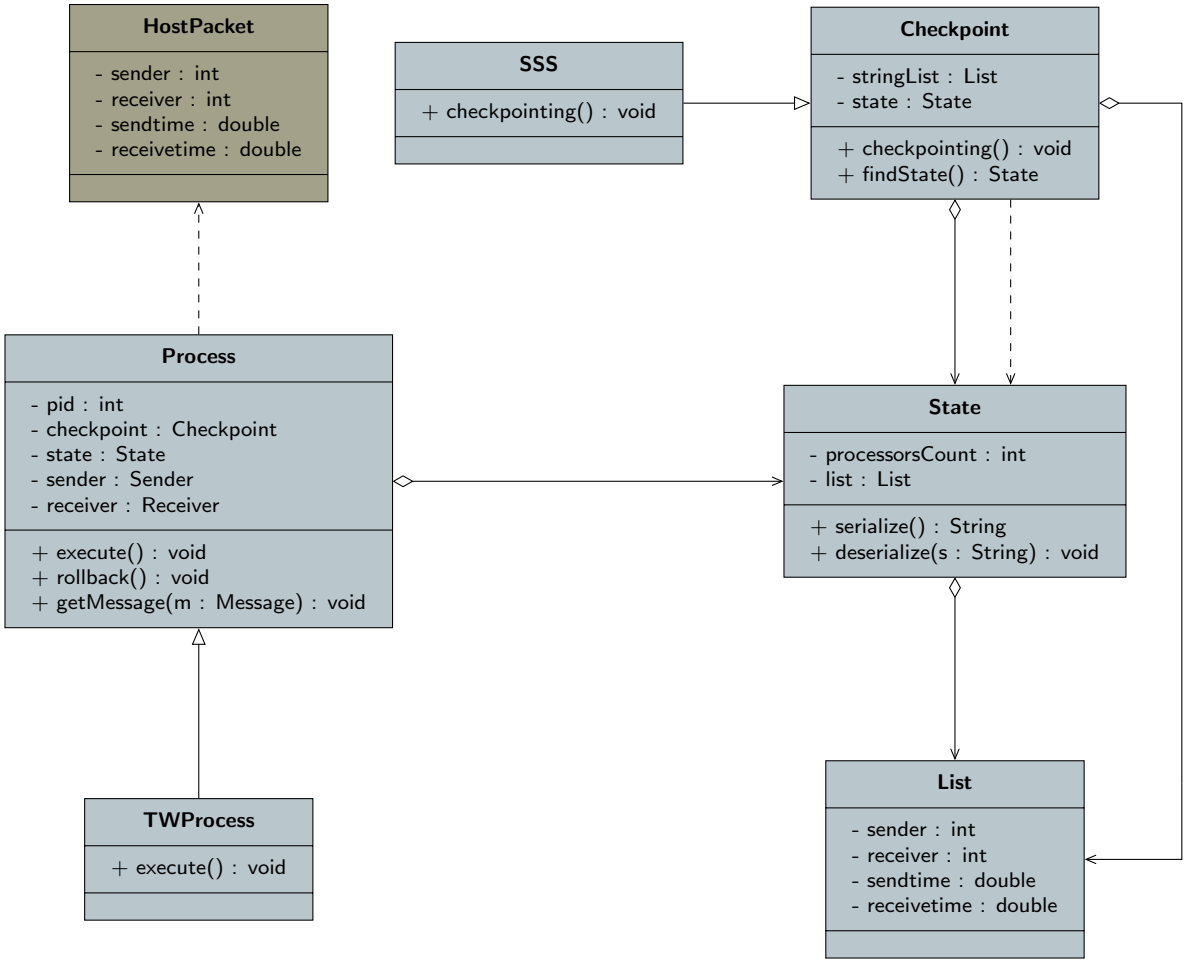


Figura 15: Diagrama do protocolo TimeWarp adaptado ao NetworkCloudSim

Como as mensagens enviadas são constantemente copiadas para se manter um estado anterior, a classe **HostPacket** deve ser do tipo **Cloneable**, além de possuir métodos para manipular o sinal, no caso do protocolo *TimeWarp*. Na versão original do *NetworkCloudSim*, os passos da simulação não eram breves o suficiente para permitir uma simulação mais realista da troca de mensagens, causando situações em que as mensagens não eram entregues no momento em que deveriam. Para isso, foi alterada a classe **Datacenter**, que possui o valor mínimo de cada passo e a classe **CloudSim**, que possui o valor do tempo mínimo entre eventos (**minTimeBetweenEvents**). Alterações foram também feitas em **NetworkHost**, pois era necessário que o simulador diferenciasse pacotes enviados localmente e entre *hosts*. A última alteração importante realizada foi a principal desse trabalho, que é o controle do atraso gerado pela rede. Originalmente, um atraso era calculado, mas baseado somente na topologia da rede, gerando valores baixos e estáveis em todos os casos. Para alterar esse comportamento, foi necessário criar um novo método na classe *CloudSim*, pois o método *send* original não deve ser alterado para essa finalidade, por ser usado também para operações internas do *NetworkCloudSim*. Assim,

o novo método, chamado de *sendWithDelay* foi adicionado com o diferencial de possuir um atraso adicional, calculado pela distribuição apresentada previamente nesse trabalho. O método pode ser observado na Listagem 1, que mostra o cálculo do atraso e a criação do evento de envio (linhas 6 e 7) que será usado pelo *NetworkCloudSim* pela manipulação da lista **future**.

Listagem 1: Método para envio com atraso

```
1 public static void sendWithDelay(int src, int dest, double delay, int
  ↳ tag, Object data) {
2     if (delay < 0) {
3         throw new IllegalArgumentException("Send delay can't be
  ↳negative.");
4     }
5
6     double cloudDelay = 10 * (1/(640 + random.nextGaussian()*121.6));
7     SimEvent e = new SimEvent(SimEvent.SEND, clock + delay +
  ↳cloudDelay, src, dest, tag, data);
8     future.addEvent(e);
9 }
```

Após as alterações mencionadas, é preciso estender e implementar uma classe para o *broker*, uma do tipo *AppCloudlet* e a classe principal com o método *main*.

Na classe derivada de *AppCloudlet*, a principal diferença para o modelo sugerido anteriormente, é a ausência de diversos estágios. Como já mencionado, não é necessário que os estágios sejam previamente definidos, deixando apenas um estágio inicial para cada *cloudlet*, que possui o tempo limite de execução para a simulação. Além disso, as *threads* que simulam os processos são iniciadas também nessa classe. O método principal dessa classe (**createCloudletList**) é apresentado na Listagem 2.

Listagem 2: Criação das cloudlets e threads

```
1 @Override
2 public void createCloudletList(List<Integer> vmIdList) {
3     int stageID = 0;
4     for(int i = 0; i < numbervm; i++) {
5         // o modelo de utilizacao dos recursos
6         UtilizationModel utilizationModel = new UtilizationModelFull
  ↳();
7
```

```
8           // new NetworkCloudlet(id do cloudlet, tempo de execucao,
    ↳quantidade de PEs, tamanho de arquivo antes da execucao, tamanho de
    ↳ arquivo depois da execucao, tamanho da memoria, modelo de
    ↳utilizacao da CPU, modelo de utilizacao da RAM, modelo de
    ↳utilizacao da largura de banda)
9           NetworkCloudlet cloudlet = new NetworkCloudlet(
    ↳NetworkConstants.currentCloudletId++, this.exeTime, 2, 256, 256,
    ↳1024, utilizationModel, utilizationModel, utilizationModel);
10          cloudlet.setUserId(userId);
11          // momento em que a cloudlet foi submetida
12          cloudlet.submittime=CloudSim.clock();
13          // inicia o numero de estagio
14          cloudlet.currStagenum=-1;
15          cloudlet.setVmId(vmIdList.get(i));
16          /* adiciona um estagio inicial de execucao para todos os
    ↳cloudlets
17           * cloudlet.stages - lista de estagios do cloudlet
18           * new TaskStage(tipo do estagio, dado, tempo de execucao, id
    ↳ do estagio, memoria utilizada, id da VM, id do cloudlet) */
19          cloudlet.stages.add(new TaskStage(NetworkConstants.EXECUTION,
    ↳ 0, 4000, stageID++, memory, vmIdList.get(0), cloudlet.
    ↳getCloudletId()));
20          stageID = 0;
21
22          // cria e inicia uma nova thread
23          if(!cl.logicalThread.isAlive() && !cl.createdThread) {
24                  cl.logicalThread.start();
25                  cl.createdThread = true;
26          }
27
28          // adiciona o cloudlet criado
29          clist.add(cloudlet);
30      }
31  }
```

Neste trecho de código (da Listagem 2), é criado o objeto que executará a tarefa (*Cloudlet*) da simulação (linha 9). Depois, é adicionado apenas um estágio para a tarefa da simulação, que será uma execução com limite máximo, nesse caso, 4000 unidades de tempo (linha 19). Entre as linhas 23 e 26, é iniciada a *thread*, que executará a simulação. Por fim, a tarefa (*Cloudlet*) é adicionada à lista **clist** (linha 29), que será utilizada posteriormente pelo *NetworkCloudSim*, para executá-la.

O *broker* possui algumas diferenças em relação ao básico apresentado pelo *NetworkCloudSim*, como dar o sinal para que as *threads* iniciem aproximadamente ao mesmo tempo, criação de identificadores sequenciais para evitar repetição, e alguns ajustes nos valores de **mips**. Após os ajustes mencionados, foram criadas as novas classes, propostas pelo *framework*.

Por último, uma nova classe, a do processo (**TWProcess**) foi adicionada ao projeto e possui o algoritmo do protocolo. Essa classe implementa **Runnable** e é usada para criar as *threads* mencionadas anteriormente. A implementação mais viável encontrada foi definir um número de passos máximo que cada processo deve executar, controlados por uma taxa de atualização também fixa para todos. O envio e recebimento de mensagens é realizado através do objeto responsável pela comunicação, com a diferença do cuidado de fazê-lo dentro de um bloco *synchronized*. O modelo de envio e recebimento de mensagens é definido e iniciado previamente para o devido uso dentro dessa *thread*. A Listagem 3 contém o método **run** do **TWProcess**. A linha 3 força o encerramento, caso esse seja requisitado. Para poder iniciar todos os *threads* aproximadamente ao mesmo tempo, é verificado o estado da variável **SimMain.startThreads** (linha 6). O controle da taxa de atualização do método é controlado entre as linhas 17 e 22, em que o algoritmo do protocolo executará 1000 vezes, passo que será incrementado a cada 4 passos do *NetworkCloudSim*, no caso desse exemplo. Entre as linhas 25 e 31, é tratado o caso da chegada de um novo cliente (ou nova mensagem) no sistema, onde um novo objeto do tipo **HostPacket** é criado e depois adicionado ordenadamente na lista de entrada. Após a chegada de uma nova mensagem, verificada na linha 36, são tratados os casos dos tipos da mensagem, sendo o caso de uma mensagem comum (não é *straggler* nem antimensagem) tratado nas linhas 42 e 43, em que a mesma é simplesmente adicionada à lista de entrada. A partir da linha 45, até a linha 54, se observa o caso da chegada de uma mensagem *straggler*, onde é obtido o estado global e as antimensagens são enviadas, seguindo a restauração do estado original (procedimento de *rollback*). Entre as linhas 57 e 60, calcula-se o GVT com a verificação da chegada de uma mensagem para cálculo de GVT (com o valor de **msgKind** igual a **HostPacket.CGVT**). Caso a mensagem possua um sinal negativo, a mesma pode ter sido recebida sem que a sua correspondente positiva tenha sido processada (linhas 66 a 68) ou já pode ter sido processada, caso em que será efetuado um *rollback*, com os passos entre as linhas 71 e 78. Após o recebimento e verificação de novas mensagens, será realizado um evento - se houver - baseado nas probabilidades definidas como já discutido anteriormente. O LVT é atualizado entre as linhas 89 e 93. Após a atualização do LVT, na linha 95, é calculado um número aleatório, que será utilizado para definir qual evento será realizado.

Os tipos de eventos possíveis são, na ordem da listagem, o processo envia uma mensagem para si - compreendido entre as linhas 100 e 104 - ou para outro processo (localizado entre as linhas 107 e 112). Quando uma mensagem é enviada para outro processo, um novo objeto do tipo **HostPacket** é instanciado e enviado, seguido da alteração do sinal da mensagem para **NEGATIVE**, para que seja adicionada à lista de saída. Por último, entre as linhas 121 e 123, é salvo o estado do processo, seguindo uma taxa de atualização de 40 passos do algoritmo.

Listagem 3: Método "run"

```
1  public void run() {
2    while(true) {
3      if(Thread.currentThread().isInterrupted())
4        break;
5      synchronized (SimMain.startThreads) {
6        if(SimMain.startThreads && (SimMain.startThreadN == creator.
   ↳getCloudletId())) {
7          System.out.println("Thread " + creator.getCloudletId() + "
   ↳iniciada");
8          SimMain.startThreadN++;
9          break;
10       }
11     }
12   }
13
14   GVT = new int[creator.getProbabilitiesVector().length-2];
15   Random rnd = new Random(System.currentTimeMillis());
16
17   while(passo < 1000) {
18     if(Thread.currentThread().isInterrupted())
19       break;
20     if (CloudSim.clock() - tempoAnterior > 4) {
21       passo++;
22       tempoAnterior = CloudSim.clock();
23
24       //chegada de um cliente
25       if(eventNumber < 1000 && creator.getProbabilitiesVector()[0] >
   ↳0) {
26         int num = rnd.nextInt(100);
27         if (num < creator.getProbabilitiesVector()[0]) {
28           LVT = SimMain.lvtEntrada;
29           SimMain.lvtEntrada+=5;
```

```
30            HostPacket temp = new HostPacket(creator.getVmId(), creator
    ↳.getVmId(), eventNumber++, CloudSim.clock(), -1, creator.
    ↳getCloudletId(), creator.getCloudletId(), LVT, LVT);
31            input.addAndSort(temp.clone());
32          }
33        }
34
35        //Verifica chegada de mensagens
36        incoming.addAll(creator.receiveMessage());
37        for(int i = 0; i < incoming.size(); i++) {
38          if(incoming.get(i).msgKind == HostPacket.NORMAL) {
39            //Mensagem normal
40            HostPacket tpkt = incoming.get(i);
41
42            if(LVT <= tpkt.getLVTreceiver()) {
43              input.addAndSort(incoming.get(i).clone());
44            } else {
45              //Mensagem straggler
46              //Envia antimensagens
47              int tamOutput = output.size() - 1;
48              globalState = checkpoint.findState(tpkt.getLVTreceiver())
    ↳;
49              sendAntiMessages(globalState, output);
50              //Restaura estado original
51              input.clear();
52              LVT = globalState.getLVT();
53              input.addAll(globalState.eventsList);
54              break;
55            }
56          } else {
57            if(incoming.get(i).msgKind == HostPacket.CGVT) {
58              //Mensagem para calculo do GVT
59              HostPacket tpkt = incoming.get(i);
60              calculateGVT(tpkt);
61            } else {
62              //Mensagem rollback
63              HostPacket tpkt = incoming.get(i);
64
65              //Mensagem ainda nao foi processada
66              if(LVT <= tpkt.getLVTreceiver()) {
67                findAndRemove(tpkt);
68              } else {
69                //Mensagem ja processada
```

```
70                //Envia antimensagens
71                int tamOutput = output.size() - 1;
72                globalState = checkpoint.findState(tpkt.getLVTreceiver
↳());
73                sendAntiMessages(globalState, output);
74                //Restaura estado original
75                input.clear();
76                LVT = globalState.getLVT();
77                input.addAll(globalState.eventsList);
78                break;
79              }
80            }
81          }
82        }
83        incoming.clear();
84        //Processar proximo evento da lista de eventos futuros
85        HostPacket tpkt = null;
86        if(input.size() > 0)
87          tpkt = input.remove(0);
88        if(tpkt != null) {
89          int serviceTime = 5;
90          if (LVT < tpkt.getLVTreceiver()) {
91            LVT = tpkt.getLVTreceiver();
92          }
93          LVT += serviceTime;
94
95          int chance = rnd.nextInt(100);
96          for (int i = 1; i < creator.getProbabilitiesVector().length ;
↳ i++) {
97            messageOut(tpkt);
98            else if (chance < creator.getProbabilitiesVector()[i]) {
99              //Gera um evento para o mesmo processo
100             if(i == creator.getCloudletId()) {
101               HostPacket temp = new HostPacket(creator.getVmId(),
↳creator.getVmId(), tpkt.data, CloudSim.clock(), -1, creator.
↳getCloudletId(), creator.getCloudletId(), LVT, LVT);
102               temp.pInicial = tpkt.pInicial;
103               temp.msgKind = HostPacket.NORMAL;
104               input.addAndSort(temp.clone());
105             } else {
106               //Gera um evento para outro processo
107               synchronized (((NetworkCloudletSpaceSharedScheduler)
↳SimMain.broker.getVmList().get(creator.getVmId()).
```

```
    ↳getCloudletScheduler()).pkttosend) {
108                 HostPacket temp = new HostPacket(creator.getVmId(),
    ↳SimMain.broker.getVmIdByCloudletId(i - 1), tpkt.data, CloudSim.
    ↳clock(), -1, creator.getCloudletId(), i - 1, LVT, LVT);
109                 temp.msgKind = HostPacket.NORMAL;
110                 creator.sendMessage(temp.clone());
111                 temp.setSignal(HostPacket.NEGATIVE);
112                 output.add(temp.clone());
113               }
114             }
115             break;
116           }
117         }
118       }
119
120       //Salvar estado global
121       if (CloudSim.clock() - tempoCheckpoint > 40) {
122         tempoCheckpoint = CloudSim.clock();
123         saveGlobalState(LVT);
124       }
125     }
126   }
127 }
```

O método *run*, da Listagem 3, é um exemplo de implementação do protocolo *TimeWarp* na classe *TWProcess* do *framework* proposto. Para que seja possível confirmar o funcionamento do simulador e do *framework*, é necessário, primeiramente, que sejam definidos modelos para teste, que serão apresentados na próxima seção.

5.3 Modelos

Os modelos que serão apresentados a seguir, foram definidos para que seja possível entender com clareza os resultados obtidos a partir do simulador projetado sobre o *NetworkCloudSim*. O primeiro, apresentado na Figura 16, ilustra um caso com 4 processos, sendo que 2 processos recebem novos eventos e direcionam o resultado do tratamento do evento a um dos outros 2 processos, que são responsáveis pela saída dos eventos do sistema. O segundo, apresentado na Figura 17, apresenta um caso que, como anteriormente, possui 2 processos recebendo novos eventos (primeiro estágio) e 2 outros responsáveis pela saída de eventos (estágio final), porém, além dos 4 processos, foram inseridos mais 4 que

receberão mensagens do primeiro estágio, para então enviar a um dos processos do estágio final.

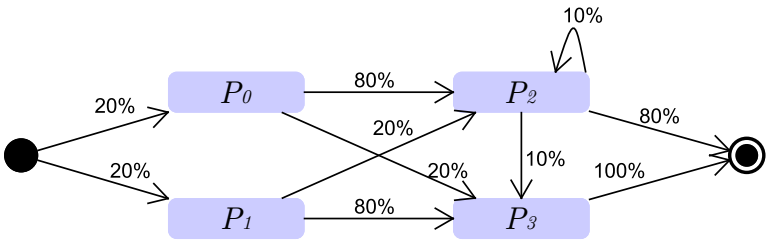


Figura 16: Diagrama de um modelo com 4 processos

Na Tabela 1 está o arquivo de entrada representando o primeiro modelo, que o programa utilizará para criar os processos e definir vetores. Os dados apresentados na Tabela 1, são referentes às probabilidades de um processo enviar uma mensagem para cada outro processo ou para que a mesma seja retirada do sistema. A estrutura desse arquivo é definida como sendo a primeira linha com o número total de processos da simulação e as linhas seguintes contendo os dados de cada processo, com 2 linhas para cada processo, contendo, na primeira linha, a probabilidade de uma nova mensagem entrar no sistema; a segunda linha deve possuir probabilidades de um processo enviar uma mensagem a outro processo - incluindo a si mesmo após o tratamento de um evento - e mais um número, referente à probabilidade da mensagem sair do sistema.

4				
20				
0	0	80	20	0
20				
0	0	20	80	0
0				
0	0	10	10	80
0				
0	0	0	0	100

Tabela 1: Dados referentes ao primeiro modelo

Sem que os modelos fossem definidos, não seria possível verificar se o algoritmo está correto ou não, pois com os dados dos modelos, já se sabe o que esperar em relação ao número de mensagens enviadas para cada processo, análise que só poderá ser feita com a obtenção de dados da simulação.

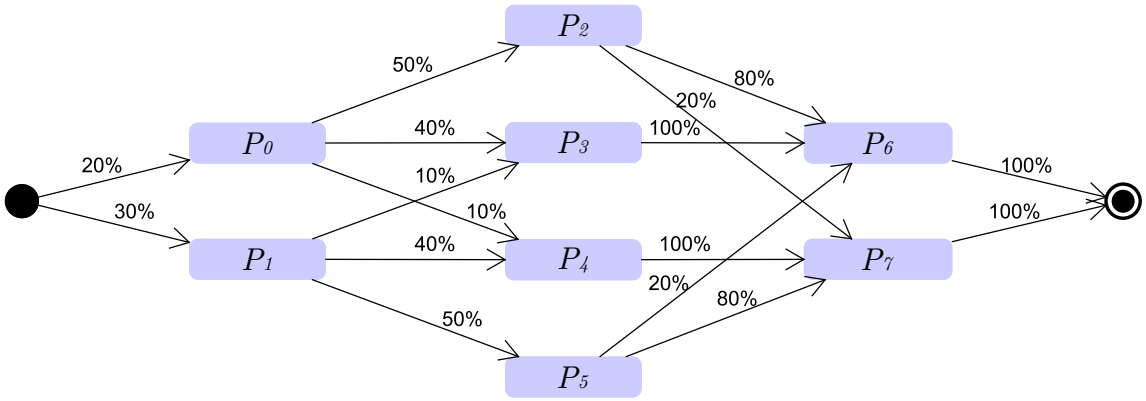


Figura 17: Diagrama de um modelo com 8 processos

5.4 Análise da consistência da implementação do protocolo

Para esse trabalho, foi definido que, para confirmar o funcionamento do protocolo, é preciso analisar se as mensagens foram enviadas de acordo com as probabilidades definidas e se as mensagens seguiram caminhos de acordo com o modelo. Dessa forma, quando a execução terminasse, seria possível utilizar esses dados, em um algoritmo que verificasse o caminho de cada mensagem até a sua saída, assim como a consistência entre os dados dos caminhos das mensagens e a contagem de envios de cada processo. Sobre os dados referentes à contagem de envios, foi proposto que, se um processo P_1 deve enviar mensagens para os processos P_3 e P_4, seguindo as probabilidades de 10% e 40%, respectivamente, no final de uma execução em que P_1 enviou 100 mensagens, a contagem de envios de P_1 para P_3 deve se aproximar de 60 e para P_4 deve se aproximar de 40. Assim sendo, para cada par de processos que se comunicam, deve-se obter, ao final da execução, dados segundo a formatação apresentada na Tabela 2.

Processo remetente (A)	Processo receptor (B)	Contagem de envios (A → B)

Tabela 2: Vetor com dados para contagem de envios entre um par de processos

Da execução da simulação, utilizando os modelos das Figuras 16 e 17 obteve-se, para os valores das estatísticas da contagem de envios, os dados apresentados na Tabela 3, para o primeiro modelo, e na Tabela 4, para o segundo modelo. Para se obter tais dados, foram feitas 100 execuções para cada modelo, com a chegada de 1000 novas mensagens em cada execução. Na primeira coluna das tabelas (3 e 4), é indicado o sentido do envio das mensagens, por exemplo, mensagens do processo P_1 para o processo P_3 são indicadas como $P_1 \rightarrow P_3$. A segunda coluna mostra, dentre as mensagens que chegou no processo, qual a porcentagem que foi enviada para o processo seguinte, por exemplo, na primeira

linha da Tabela 3, de todas as mensagens que chegaram em P_0, 79,85% foram enviadas para P_2. A terceira coluna indica o desvio padrão dos valores obtidos.

Tabela 3: Estatísticas para o primeiro modelo

Sentido do envio ($*$ ≡ saída)	**Média (%)**	**Desvio padrão**
$P_0 \rightarrow P_2$	79,85%	0.0206
$P_0 \rightarrow P_3$	20,15%	0.0206
$P_1 \rightarrow P_2$	19,75%	0.0208
$P_1 \rightarrow P_3$	80,25%	0.0208
$P_2 \rightarrow P_2$	8,88%	0.0112
$P_2 \rightarrow P_3$	8,92%	0.0127
$P_2 \rightarrow *$	82,20%	0.0166
$P_3 \rightarrow *$	100%	-

Tabela 4: Estatísticas para o segundo modelo

Sentido do envio ($*$ ≡ saída)	**Média (%)**	**Desvio padrão**
$P_0 \rightarrow P_2$	50,08%	0.0234
$P_0 \rightarrow P_3$	39,80%	0.0234
$P_0 \rightarrow P_4$	10,12%	0.0129
$P_1 \rightarrow P_3$	10,02%	0.0131
$P_1 \rightarrow P_4$	40,15%	0.0199
$P_1 \rightarrow P_5$	49,83%	0.0227
$P_2 \rightarrow P_6$	78,43%	0.0271
$P_2 \rightarrow P_7$	20,49%	0.0202
$P_3 \rightarrow P_6$	49,26%	0.0244
$P_3 \rightarrow P_7$	49,68%	0.0279
$P_4 \rightarrow P_6$	48,05%	0.0309
$P_4 \rightarrow P_7$	49,67%	0.0148
$P_5 \rightarrow P_6$	20,03%	0.0174
$P_5 \rightarrow P_7$	79,43%	0.0147
$P_6 \rightarrow *$	100%	-
$P_7 \rightarrow *$	100%	-

Além das contagens de envios, para cada envio de mensagem por um processo, foi gerado um vetor com os dados para identificar a mensagem, processo remetente e processo receptor, sendo que, na saída da mensagem, o processo receptor foi definido como -1. A Tabela 5 mostra a estrutura desse vetor.

Identificador da mensagem	Processo remetente	Processo receptor

Tabela 5: Vetor representando o envio de uma mensagem entre 2 processos

Para analisar esses dados, utilizou-se um programa para verificar a consistência, representado pelo pseudocódigo da Listagem 4, que demonstra a análise para uma mensagem. O que o pseudocódigo mostra é que, para cada número de identificação de mensagem (id), deve-se buscar a primeira mensagem com essa identificação, verificar se o remetente dessa mensagem difere do receptor anterior (implicando em um problema, pois o receptor de uma mensagem passa a ser o remetente, após o recebimento), somar 1 ao contador referente àquele par e atualizar o receptor. Esse *loop* acaba quando atinge o valor -1 de receptor, indicando que a mensagem saiu do sistema.

Listagem 4: Pseudocódigo da análise dos caminhos das mensagens

```
contador[1...N][1...N], remetente, receptor, id: numerico
receptor <- 0
id <- 0

enquanto(receptor <> -1)
{
    vetor <- buscaProximoVetorPeloId(id)
    se(vetor[2] <> receptor)
        interrompe
    remetente <- vetor[2]
    contador[remetente][receptor]++
    receptor <- vetor[3]
}
```

A importância da validação ocorre pois, se o algoritmo não está funcionando corretamente, podem ocorrer erros de análise, levando a uma situação em que um *rollback* só ocorreu por erro do algoritmo, ou ainda uma situação em que o retorno deveria ocorrer, mas nada aconteceu. A análise da ocorrência do *rollback* será realizada na seção seguinte, pela observação dos resultados do simulador após a sua execução.

5.5 Análise do histórico

A partir da estrutura de validação elaborada, foram feitos experimentos para analisar o seu funcionamento. Como se utilizou um simulador de computação em nuvem, é possível observar o dado que se desejar, sendo que os escolhidos para esse trabalho foram:

- Evento de envio, com os identificadores do processo receptor e emissor, LVT e identificador da mensagem.

- Evento de recebimento, com os identificadores do processo receptor e emissor, LVT, identificador da mensagem, relógio (para a comparação com o atraso) e o sinal (verdadeiro ou falso).
- Atraso.
- Saída da mensagem.
- *Rollback*, com informações do processo que está realizando o retorno e o LVT antes do retorno.

Esses dados são apresentados em um arquivo de *log*, que contém todos os eventos e informações de todos os processos. Cada linha desse arquivo, que contém as informações da execução, pode ser dos tipos apresentados na Figura 18 e serão explicados a seguir:

- A linha 1, representa um evento de envio, ou seja, o processo **A** enviou uma mensagem, com o número de identificação **ID**, para o processo **B**, em um determinado **tempo lógico** (LVT);
- o padrão da linha 2, indica que a entrega da mensagem, correspondente ao envio imediatamente acima dessa linha, será entregue com o **tempo** de atraso informado;
- na linha 3, demonstra-se um evento de recebimento, indicando que o processo **A** recebeu uma mensagem com um **sinal** e número de identificação **ID**, em um determinado tempo da simulação do **NetworkCloudSim** e com LVT igual a um **tempo lógico**;
- a linha 4 é utilizada para quando uma mensagem com identificador **ID** sai do sistema;
- finalmente, a linha 5, mostra que houve um *rollback* no processo **P**, em um **tempo lógico**.

Um exemplo de um momento da simulação utilizando o segundo modelo (Figura 17) é observado na Figura 19, que contém um trecho do arquivo de *log* da simulação (executada a partir da adaptação do *NetworkCloudSim*). Nele, pode-se acompanhar o momento do envio do dado (identificador da mensagem) 1147 - localizado na linha 1 - do processo 1 para o processo 4, até o seu recebimento (linha 16). As linhas de 2 a 15 demonstram os eventos e atrasos que ocorreram entre o envio e recebimento da mensagem analisada.

Figura 18: Informações escolhidas para o arquivo de *log*

```
1 Envio> de A para B, dado: ID, LVT: tempo lógico
2 Atraso> tempo de atraso
3 Recebimento> em A de B, dado: ID, LVT: tempo lógico, relogio: reló
  ↳gio do NetworkCloudSim, sinal: sinal da mensagem
4 Saida> ID
5 Rollback> de P, LVT: tempo lógico
```

Figura 19: Momento de uma simulação com os eventos e atrasos apresentados como saída

```
1  Envio> de 1 para 4, dado: 1147, LVT: 10425
2  Atraso> 7.843
3  Saida> dado: 1142
4  Envio> de 0 para 2, dado: 937, LVT: 10430
5  Atraso> 9.412
6  Recebimento> em 0 de 3, dado: 936, LVT: 10420, relogio: 10495.534,
    ↳ sinal: positivo
7  Envio> de 3 para 7, dado: 936, LVT: 10425
8  Atraso> 8.472
9  Recebimento> em 4 de 1, dado: 1146, LVT: 10415, relogio:
    ↳10495.814, sinal: positivo
10 Envio> de 4 para 6, dado: 1146, LVT: 10420
11 Atraso> 7.903
12 Envio> de 1 para 5, dado: 1148, LVT: 10435
13 Atraso> 5.811
14 Envio> de 0 para 2, dado: 938, LVT: 10440
15 Atraso> 6.534
16 Recebimento> em 4 de 1, dado: 1147, LVT: 10425, relogio:
    ↳10503.815, sinal: positivo
```

É possível observar também o momento em que um *rollback* ocorreu por conta de um atraso, na Figura 20. Pode-se observar esse fato tanto pelo atraso, que no envio do processo 5 (linha 3) foi quase o dobro do envio do processo 2 (linha 5), como pelo fato de que o envio do processo 2 (linha 5) ocorreu após o envio do processo 5 (linha 2), mas o recebimento em 6 da mensagem do processo 2 (linha 8) ocorreu antes do recebimento da mensagem do processo 5 (linha 10), fazendo com que o LVT do processo 6 fosse atualizado pela mensagem do processo 2.

Figura 20: Momento de um *rollback*

```
1  Recebimento> em 5 de 1, dado: 686, LVT: 6275, relogio: 6428.649,
    ↳sinal: positivo
2  Envio> de 5 para 6, dado: 686, LVT: 6280
3  Atraso> 17.718
4  Recebimento> em 2 de 0, dado: 568, LVT: 6280, relogio: 6428.784,
    ↳sinal: positivo
5  Envio> de 2 para 6, dado: 568, LVT: 6285
6  Atraso> 9.069
7  ...
8  Recebimento> em 6 de 2, dado: 568, LVT: 6285, relogio: 6438.916,
    ↳sinal: positivo
9  ...
10 Recebimento> em 6 de 5, dado: 686, LVT: 6280, relogio: 6446.916,
    ↳sinal: positivo
11 Rollback> de 6, LVT: 6290
```

5.6 Considerações finais

Neste capítulo foi apresentado o *framework NetworkCloudSim* e foram propostas alterações em sua estrutura de classes, baseadas no *framework* definido por Cruz (2009). Embora o *NetworkCloudSim* possua documentação aberta, detalhes de sua implementação que eram problemáticos para o caso apresentado geraram diversos problemas nos testes de implementação das alterações propostas. Entre os casos problemáticos, pode-se citar o tempo do passo da simulação, que precisou ser encurtado, para evitar, por exemplo, que duas mensagens enviadas em tempos (lógico e de simulação) distintos fossem recebidas simultaneamente. É possível citar também que, entre as primitivas (*EXECUTION*, *WAIT_SEND* e *WAIT_RECV*) disponibilizadas pelo *NetworkCloudSim*, apenas a *EXECUTION* foi necessária, pois não seria possível definir previamente a sequência de enviar/receber, levando em conta que os *rollbacks* são imprevisíveis e isso geraria a necessidade da reconfiguração de toda a sequência de envio e recebimento de mensagens. Outro caso foi o envio das mensagens de forma "global" e "local", disponibilizados no *NetworkCloudSim*, em que as mensagens jamais eram enviadas de forma "global", evitando que fossem repassadas entre os *switches*.

Esse trabalho, inicialmente, pretendia gerar a sequência de enviar/receber completa, indicando os locais onde ocorreram os *rollbacks*, caso que, como mencionado, gerava problemas desnecessários de reorganização dos estágios da simulação. Mesmo após identificar esse erro e o corrigir, o fato de que não havia *threads* ou processos, reais ou não, sem-

pre era seguido da mesma sequência de "processos" enviando e recebendo as mensagens. Isso gerou mais um problema, pois uma mensagem que era enviada de um processo para outro que já havia realizado suas ações de recebimento em um dado passo da simulação, provocava em alguns momentos um falso atraso. Esse atraso ocorria pois o tempo lógico do recebimento era inevitavelmente atrasado, também por conta do tamanho do passo da simulação, que ainda não havia sido diminuído. Por último, com o passo de simulação diminuído e com *threads* executando "individualmente", tais problemas foram contornados, embora a implementação de um protocolo de sincronização nesse ambiente simulado proposto ainda não esteja completa.

É preciso lembrar que, com a previsão do impacto dos atrasos, é possível propor melhorias nos algoritmos, mas não é possível prever o custo do serviço de nuvem (geralmente pagos sob demanda). Prever o custo é difícil, pois os tempos apresentados na simulação podem não representar o tempo da simulação distribuída em si, não só pela possibilidade das diferenças dos modelos, mas também pelo fato de tais ambientes possuírem atrasos de processamento e acesso a memória que são ainda impossíveis de equacionar. Esse problema ao equacionar acontece devido à arquitetura da máquina em que ocorreram os estudos e a arquitetura do ambiente em nuvem, que podem ser diferentes. Outro problema é que até mesmo duas arquiteturas do mesmo serviço podem ser diferentes entre si.

6 Conclusão

O objetivo desse trabalho foi desenvolver um sistema que permitisse a análise, passo a passo, da execução de uma simulação distribuída, em um ambiente de nuvem simulado. Com o estudo de alguns trabalhos sobre desempenho e estabilidade em nuvem, como o de Schad, Dittrich e Quiané-Ruiz (2010), CloudSpectator (2012), Dejun, Pierre e Chi (2009) e Wang e Ng (2010), ficou evidente que existe uma grande flutuação no desempenho da transmissão em rede, processamento e acesso a memória e disco, mesmo quando as VMs se comunicam dentro de uma mesma área geográfica e entre computadores com a mesma arquitetura. A partir desses resultados, criou-se um modelo teórico e desenvolveu-se uma extensão ao *NetworkCloudSim*, adaptando o *framework* apresentado em Cruz (2009). Essa extensão permite a programação de protocolos otimistas, no qual o protocolo *Time Warp* foi implementado como exemplo, apresentando resultados consistentes com os valores esperados dos testes escolhidos. Essa extensão inclui a possibilidade de se criar atrasos para envio de mensagens, sendo que esses atrasos são gerados a partir de uma distribuição normal, criando uma instabilidade semelhante à real, descrita por Schad, Dittrich e Quiané-Ruiz (2010).

A escolha do *NetworkCloudSim* não foi feita imediatamente, uma vez que o projeto começou com a análise de como seriam executados os protocolos *Time Warp* e *Rollback Solidário* em um ambiente em nuvem. Para a execução, foi criado um serviço privado de computação em nuvem utilizando o *software* OpenStack. Ao se observar que o problema não estava na implementação para um serviço de nuvem, foram estudados os casos de implementação de outros autores, sendo que, em Malik, Park e Fujimoto (2010), foi apontado o problema do aumento do número de *rollbacks* na execução do protocolo *Time Warp* em serviços públicos de nuvem. A partir dessa informação, foram feitas análises de estabilidade no serviço privado implementado (utilizando o OpenStack), que por não possuir diversos clientes concorrendo pelos mesmos recursos, não apresentava a instabilidade necessária para a confirmação do seu impacto na ocorrência de *rollbacks*.

Para se conseguir um sistema propositadamente instável, foram feitas algumas ten-

tativas de se gerar um atraso no envio do protocolo, mas por atrapalharem o seu funcionamento, começou-se a pesquisar ambientes simulados de computação em nuvem. O *NetworkCloudSim* foi escolhido por já possuir as primitivas de envio e recebimento de mensagens e o atraso simulado foi implementado em primeiro lugar. Ao se tentar implementar o comportamento de um protocolo otimista no *NetworkCloudSim*, encontraram-se diversos obstáculos, sendo que o primeiro foi a ausência de uma primitiva de recebimento não bloqueante, fazendo com que uma mensagem atrasada fosse responsável por segurar o sistema inteiro. Ao se resolver esse problema, surgiu um novo contratempo, pois não se podia prever a sequência de envios e recebimentos previamente, e a geração em tempo de execução não é suportada nativamente pelo *software* escolhido. Quando se decidiu enviar e receber as mensagens diretamente, sem o uso das primitivas oferecidas, a implementação do protocolo apresentou diversas inconsistências, por ser uma tentativa de se imitar o comportamento de um programa paralelo em um sistema sequencial. Finalmente, todo o protocolo foi replanejado e foi criada uma nova classe para executar uma *thread*, versão que foi mantida até a apresentação desse trabalho.

A decisão por se simular o protocolo em um simulador de computação em nuvem se mostrou correta, pois além de se controlar a instabilidade, também é possível analisar todo o histórico de envios, recebimentos, atrasos e *rollbacks*. Essa análise permite que, ao se observar um *rollback* no arquivo de *log*, sejam desenvolvidas soluções para evitar o caso observado através de modificações no protocolo otimista original. A possibilidade de se controlar e analisar os resultados no simulador pode agilizar o processo de implementação das alterações que visam a melhoria do desempenho dos protocolos otimistas em serviços públicos de nuvem, além de eliminar a necessidade de se pagar por um serviço como o *Amazon EC2*, quando se busca apenas a otimização do protocolo.

Foi analisada também a possibilidade de se criar um serviço privado de computação em nuvem utilizando *softwares* como CloudStack e OpenStack. Nesse caso, a vantagem da possível economia com estrutura física foi abandonada, focando apenas em vantagens de se criar um serviço de simulação, que se mostrou viável, pois com uma nuvem própria é possível gerenciar todos os aspectos de *hardware* e *software*, além de possibilitar um novo tipo de migração, a migração de VM entre as máquinas do *cluster*. A principal vantagem dessa abordagem, seria a estabilidade - mesmo com o desempenho prejudicado pela natureza da virtualização - e a mobilidade, permitindo que uma simulação seja observada, iniciada ou projetada, por exemplo, de um aparelho móvel.

Um ponto que se deve melhorar no sistema é a estabilidade do *Time Warp*, uma vez

que foi implementada uma versão simplificada desse protocolo. Essa nova implementação possibilitaria que o protocolo fosse validado de outras formas, permitindo a sua comparação com a execução de um caso real de simulação distribuída. Outra possibilidade para trabalho futuro, é a conclusão da implementação do *Rollback Solidário*, uma vez que o mesmo não foi finalizado até a data da apresentação desse trabalho. Diversas novas funcionalidades podem ser adicionadas, como o desenvolvimento de uma interface de navegador que seja familiar aos ambientes em nuvem reais, demonstrando a quantidade de máquinas virtuais criadas, uso dos recursos e um terminal para a interação com o sistema. Outro recurso que poderá ser adicionado a uma futura implementação de uma interface de navegador é a visualização de gráficos mostrando os resultados estatísticos da simulação, tanto em tempo real como os resultados finais. Outra possibilidade para adição ao sistema é a adaptação de novas classes de protocolos de simulação ou aplicações baseadas em envio e recebimento de mensagens.

Referências

ARMBRUST, M. et al. *Above the Clouds: A Berkeley View of Cloud Computing*. [S.l.], 2009.

AYMERICH, F. M.; FENU, G.; SURCIS, S. An approach to a Cloud Computing network. In: *Applications of Digital Information and Web Technologies, 2008. ICADIWT 2008. First International Conference on the*. [s.n.], 2008. p. 113–118. Disponível em: <http://dx.doi.org/10.1109/icadiwt.2008.4664329>.

BABAOGLU, O.; MARZULLO, K. Consistent global states of distributed systems: Fundamental concepts and mechanisms. *Distributed Systems*, Citeseer, v. 2, n. January, p. 55–96, 1993.

BANKS, J. et al. *Discrete-event System Simulation*. [S.l.: s.n.], 2009.

BAUER, P. et al. Efficient inter-process synchronization for parallel discrete event simulation on multicores. In: *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. New York, NY, USA: ACM, 2015. (SIGSIM PADS '15), p. 183–194. ISBN 978-1-4503-3583-6. Disponível em: <http://doi.acm.org/10.1145/2769458.2769476>.

BOHN, R. et al. NIST cloud computing reference architecture. In: *Services (SERVICES), 2011 IEEE World Congress on*. [S.l.: s.n.], 2011. p. 594–596.

BUYYA, R.; RANJAN, R.; CALHEIROS, R. N. Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. *CoRR*, abs/0907.4878, 2009. Disponível em: <http://arxiv.org/abs/0907.4878>.

BUYYA, R. et al. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 25, n. 6, p. 599–616, jun. 2009. ISSN 0167-739X. Disponível em: <http://dx.doi.org/10.1016/j.future.2008.12.001>.

CARAGNANO, G. et al. Scalability of a parallel application in hybrid cloud. In: *Proceedings of the 2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems*. Washington, DC, USA: IEEE Computer Society, 2014. (CISIS '14), p. 451–456. ISBN 978-1-4799-4325-8. Disponível em: <http://dx.doi.org/10.1109/CISIS.2014.64>.

CHEN, L.-l. et al. A well-balanced time warp system on multi-core environments. In: *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*. Washington, DC, USA: IEEE Computer Society, 2011. (PADS '11), p. 1–9. ISBN 978-1-4577-1363-7. Disponível em: <http://dx.doi.org/10.1109/PADS.2011.5936752>.

CHHABRA, S.; DIXIT, V. S. Cloud computing: State of the art and security issues. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 40, n. 2, p. 1–11, abr. 2015. ISSN 0163-5948. Disponível em: <http://doi.acm.org/10.1145/2735399.2735405>.

CHILD, R.; WILSEY, P. Dynamically adjusting core frequencies to accelerate time warp simulations in many-core processors. In: *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*. Washington, DC, USA: IEEE Computer Society, 2012. (PADS '12), p. 35–43. ISBN 978-0-7695-4714-5. Disponível em: <http://dx.doi.org/10.1109/PADS.2012.15>.

CHILD, R.; WILSEY, P. A. Using dvfs to optimize time warp simulations. In: *Proceedings of the Winter Simulation Conference*. Winter Simulation Conference, 2012. (WSC '12), p. 288:1–288:12. Disponível em: <http://dl.acm.org/citation.cfm?id=2429759.2430148>.

CHWIF, L.; MEDINA, A. *Modelagem e Simulação de Eventos Discretos*. [S.l.]: LEONARDO CHWIF, 2014. ISBN 9788535279320.

CLOUDSIM. *CloudSim*. 2015. Disponível em: <http://www.cloudbus.org/cloudsim/>.

CLOUDSPECTATOR, T. *Cloud performance and stability report: Amazon EC2*. [S.l.], 2012.

CLOUDSTACK. *CloudStack*. 2015. Disponível em: <http://cloudstack.apache.org>.

CRUZ, L. *Projeto de um Framework para o Desenvolvimento de Aplicações de Simulação Distribuída*. Dissertação (Mestrado) — Universidade Federal de Itajubá, Itajubá – MG, 2009.

D'ANGELO, G.; MARZOLLA, M. New trends in parallel and distributed simulation: From many-cores to cloud computing. *Simulation Modelling Practice and Theory*, v. 49, n. 0, p. 320 – 335, 2014. ISSN 1569-190X. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1569190X14001014>.

DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In: *OSDI'04: PROCEEDINGS OF THE 6TH CONFERENCE ON SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION*. [S.l.]: USENIX Association, 2004.

DEJUN, J.; PIERRE, G.; CHI, C.-H. EC2 performance analysis for resource provisioning of service-oriented applications. In: *Proceedings of the 2009 International Conference on Service-oriented Computing*. Berlin, Heidelberg: Springer-Verlag, 2009. (ICSOC/ServiceWave'09), p. 197–207. ISBN 3-642-16131-6, 978-3-642-16131-5. Disponível em: <http://dl.acm.org/citation.cfm?id=1926618.1926641>.

DICKMAN, T.; GUPTA, S.; WILSEY, P. A. Event pool structures for PDES on many-core beowulf clusters. In: *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. New York, NY, USA: ACM, 2013. (SIGSIM PADS '13), p. 103–114. ISBN 978-1-4503-1920-1. Disponível em: <http://doi.acm.org/10.1145/2486092.2486106>.

DIEGUES, N.; ROMANO, P. Time-warp: Efficient abort reduction in transactional memory. *ACM Trans. Parallel Comput.*, ACM, New York, NY, USA, v. 2, n. 2, p. 12:1–12:44, jun. 2015. ISSN 2329-4949. Disponível em: <http://doi.acm.org/10.1145/2775435>.

DRYAD. *Dryad*. 2015. Disponível em: <http://research.microsoft.com/en-us/projects/dryad/>.

EC2, A. E. C. C. *Amazon EC2*. 2015. Disponível em: <http://aws.amazon.com/pt/ec2/>.

EKANAYAKE, J. et al. *High Performance Parallel Computing with Cloud and Cloud Technologies*. 2009.

ELASTICHOSTS. *ElasticHosts*. 2015. Disponível em: <http://www.elastichosts.com/>.

ERL, T.; PUTTINI, R.; MAHMOOD, Z. *Cloud Computing: Concepts, Technology & Architecture*. 1st. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2013. ISBN 0133387526, 9780133387520.

EVANGELINOS, C.; HILL, C. N. Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In: CLOUD COMPUTING AND ITS APPLICATIONS. 2008. Disponível em: <http://www.cca08.org/speakers/evangelinos.php>.

FUJIMOTO, R. M.; MALIK, A. W.; PARK, A. J. Parallel and distributed simulation in the cloud. jul. 2010.

GABRIELSOON, J. et al. *Cloud Computing in telecommunications*. jan. 2010.

GARG, S. K.; BUYYA, R. Networkcloudsim: Modelling parallel applications in cloud simulations. In: *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*. Washington, DC, USA: IEEE Computer Society, 2011. (UCC '11), p. 105–113. ISBN 978-0-7695-4592-9. Disponível em: <http://dx.doi.org/10.1109/UCC.2011.24>.

GOGRID. *GoGrid*. 2015. Disponível em: <http://www.gogrid.com/>.

GUAN, S. *A Multi-layered Scheme for Distributed Simulations on the Cloud Environment*. Dissertação (Mestrado), 2015.

HADOOP, A. *Apache Hadoop*. 2015. Disponível em: <http://hadoop.apache.org/>.

HE, H. et al. An efficient and secure cloud-based distributed simulation system. ago. 2012.

HUANG, W. et al. The state of public infrastructure-as-a-service cloud security. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 47, n. 4, p. 68:1–68:31, jun. 2015. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/2767181>.

IEEE STANDARD. IEEE standard for modeling and simulation (M & S) High Level Architecture (HLA)– framework and rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, p. 1–38, Aug 2010.

JADEJA, Y.; MODI, K. Cloud computing - concepts, architecture and challenges. In: *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on*. [S.l.: s.n.], 2012. p. 877–880.

JAFER, S.; LIU, Q.; WAINER, G. Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory*, v. 30, n. 0, p. 54 – 73, 2013. ISSN 1569-190X. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1569190X12001244>.

JEFFERSON, D. R. Virtual time. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 7, n. 3, p. 404–425, jul. 1985. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/3916.3988>.

JUVE, G. et al. Scientific workflow applications on amazon EC2. In: *E-Science Workshops, 2009 5th IEEE International Conference on*. [S.l.: s.n.], 2009. p. 59–66.

KIM, W. Cloud architecture: A preliminary look. In: *Proceedings of the 9th International Conference on Advances in Mobile Computing and Multimedia*. New York, NY, USA: ACM, 2011. (MoMM '11), p. 2–6. ISBN 978-1-4503-0785-7. Disponível em: <http://doi.acm.org/10.1145/2095697.2095699>.

KSHEMKALYANI, A. D.; SINGHAL, M. *Distributed Computing: Principles, Algorithms, and Systems*. 1. ed. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521876346.

KVM. *KVM Project*. 2015. Disponível em: <http://www.linux-kvm.org/page/Main_Page>.

LAW, A.; KELTON, W. *Simulation modeling and analysis*. McGraw-Hill, 2000. (McGraw-Hill series in industrial engineering and management science). ISBN 9780070592926. Disponível em: <http://books.google.com.br/books?id=QqkZAQAAIAAJ>.

LEDYAYEV, R.; RICHTER, H. High performance computing in a cloud using openstack. In: . [S.l.: s.n.], 2014.

LENK, A. et al. What are you paying for? performance benchmarking for infrastructure-as-a-service offerings. In: *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. [S.l.: s.n.], 2011. p. 484–491. ISSN 2159-6182.

LIU, Y.; VLASSOV, V.; NAVARRO, L. Towards a community cloud storage. In: *Proceedings of the 2014 IEEE 28th International Conference on Advanced Information Networking and Applications*. Washington, DC, USA: IEEE Computer Society, 2014. (AINA '14), p. 837–844. ISBN 978-1-4799-3630-4. Disponível em: <http://dx.doi.org/10.1109/AINA.2014.102>.

MALIK, A. W.; PARK, A. J.; FUJIMOTO, R. M. An optimistic parallel simulation protocol for cloud computing environments. out. 2010.

MANCINI, E. et al. Simulation in the cloud using handheld devices. In: *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. [S.l.: s.n.], 2012. p. 867–872.

MELL, P.; GRANCE, T. *The NIST Definition of Cloud Computing*. 26. ed. [S.l.], jul. 2009. Disponível em: <http://www.csrc.nist.gov/groups/SNS/cloud-computing/>.

MOLLAH, M.; ISLAM, K.; ISLAM, S. Next generation of computing through cloud computing technology. In: *Electrical Computer Engineering (CCECE), 2012 25th IEEE Canadian Conference on*. [S.l.: s.n.], 2012. p. 1–6. ISSN 0840-7789.

MOREIRA, E. *Rollback Solidário: um novo protocolo otimista para simulação distribuída*. Tese (Doutorado) — Universidade de São Paulo, São Carlos – SP, 2005. Disponível em: <http://books.google.com.br/books?id=DjZvGwAACAAJ>.

OPENSTACK. *OpenStack*. 2015. Disponível em: <http://www.openstack.org>.

OSTERMANN, S. et al. A performance analysis of EC2 cloud computing services for scientific computing. In: AVRESKY, D. et al. (Ed.). *Cloud Computing*. Springer Berlin Heidelberg, 2010, (Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, v. 34). p. 115–131. ISBN 978-3-642-12635-2. Disponível em: <http://dx.doi.org/10.1007/978-3-642-12636-9_9>.

PELLEGRINI, A.; QUAGLIA, F. NUMA time warp. In: *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. New York, NY, USA: ACM, 2015. (SIGSIM PADS '15), p. 59–70. ISBN 978-1-4503-3583-6. Disponível em: <http://doi.acm.org/10.1145/2769458.2769479>.

PIENTA, R. S.; FUJIMOTO, R. M. On the parallel simulation of scale-free networks. In: *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. New York, NY, USA: ACM, 2013. (SIGSIM PADS '13), p. 179–188. ISBN 978-1-4503-1920-1. Disponível em: <http://doi.acm.org/10.1145/2486092.2486115>.

ROSADO, T.; BERNARDINO, J. An overview of openstack architecture. In: *Proceedings of the 18th International Database Engineering & Applications Symposium*. New York, NY, USA: ACM, 2014. (IDEAS '14), p. 366–367. ISBN 978-1-4503-2627-8. Disponível em: <http://doi.acm.org/10.1145/2628194.2628195>.

ROSENBLUM, M.; GARFINKEL, T. Virtual machine monitors: current technology and future trends. *Computer*, v. 38, n. 5, p. 39–47, 2005. ISSN 0018-9162.

SCHAD, J.; DITTRICH, J.; QUIANÉ-RUIZ, J.-A. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, VLDB Endowment, v. 3, n. 1-2, p. 460–471, set. 2010. ISSN 2150-8097. Disponível em: <http://dl.acm.org/citation.cfm?id=1920841.1920902>.

SIMZENTRUM. *A Cloudbased Software Infrastructure for Distributed Simulation*. 2015. Disponível em: <https://www.simzentrum.de/en/projekte/cloudbasiertesoftware-infrastrukturfuerverteiltesimulation/>.

SINGH, V. P. *System Modeling and Simulation*. [S.l.]: New Age International (P) Limited, 2009. ISBN 9788122423860.

SMITH, J.; NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN 1558609105.

TAY, S. C.; TEO, Y. M. Probabilistic checkpointing in time warp parallel simulation. In: *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on*. [S.l.: s.n.], 2000. p. 366–373. ISSN 1526-7539.

VANMECHELEN, K.; MUNCK, S. D.; BROECKHOVE, J. Conservative distributed discrete event simulation on amazon EC2. In: *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. [S.l.: s.n.], 2012. p. 853–860.

VCLOUD. *VMware vCloud Suite*. 2015. Disponível em: <http://www.vmware.com/br/products/vcloud-suite>.

VEE, V.-Y.; HSU, W.-J. Pal:a new fossil collector for time warp. In: *Parallel and Distributed Simulation, 2002. Proceedings. 16th Workshop on*. [S.l.: s.n.], 2002. p. 31–38. ISSN 1087-4097.

VMWARE. *VMWare*. 2015. Disponível em: <http://www.vmware.com/br>.

WALKER, E. Benchmarking amazon EC2 for high-performance scientific computing. *Usenix Login*, v. 33, n. 5, p. 18–23, 2008.

WANG, G.; NG, T. S. E. The impact of virtualization on network performance of amazon EC2 data center. In: *Proceedings of the 29th Conference on Information Communications*. Piscataway, NJ, USA: IEEE Press, 2010. (INFOCOM'10), p. 1163–1171. ISBN 978-1-4244-5836-3. Disponível em: <http://dl.acm.org/citation.cfm?id=1833515.1833691>.

XENSERVER. *XenServer*. 2015. Disponível em: <http://www.xenserver.org/>.

ZOU, T. et al. Making time-stepped applications tick in the cloud. In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. New York, NY, USA: ACM, 2011. (SOCC '11), p. 20:1–20:14. ISBN 978-1-4503-0976-9. Disponível em: <http://doi.acm.org/10.1145/2038916.2038936>.

APÊNDICE A – Utilização do NetworkCloudSim

Nesse apêndice, será demonstrada como é feita a configuração do *NetworkCloudSim*, mostrando a criação dos objetos do *Datacenter*, como o elemento de processamento (*Pe*), máquinas hospedeiras (*Host*) e os *Switches*.

Primeiramente é criado um objeto para representar a unidade de CPU, medida em MIPS. Esse objeto é chamado de elemento de processamento (*Pe*) e sua criação é ilustrada na listagem 1, onde é criada uma lista, em que mais de um *Pe* pode ser adicionado.

Listagem 1: Criação do Pe

```
int mips = 10;

List<Pe> peList = new ArrayList<Pe>();

// new Pe(id, new PeProvisionerSimple(mips))
peList.add(new Pe(0, new PeProvisionerSimple(mips)));
```

Após a criação do *Pe*, deve-se criar o *Host*, que possui **RAM**, largura de banda (**BW**), armazenamento (**storage**) e a lista de *Pe* (**peList**), ou seja, é uma representação de uma máquina do centro de dados. O exemplo da criação de um *host* pode ser observada na listagem 2.

Listagem 2: Criação do *Host*

```
int ram = 2048;
long storage = 10000;
int bw = 1000;

// new NetworkHost(id, Ram Provisioner, Bw Provisioner, storage,
  ↳lista de Pes, escalonador de maquina virtual)
List<NetworkHost> hostList = new ArrayList<NetworkHost>();
hostList.add(new NetworkHost(
```

```
                        0,
                        new RamProvisionerSimple(ram),
                        new BwProvisionerSimple(bw),
                        storage,
                        peList,
                        new VmSchedulerTimeShared(peList)));
```

Com os *hosts* criados, deve-se criar o centro de dados (*datacenter*), mas antes se instancia um objeto do tipo *DatacenterCharacteristics*, que contém todas as características do *datacenter*, conforme exemplo da listagem 3.

Listagem 3: Criação do *DatacenterCharacteristics*

```
String arch = "x86"; // arquitetura do sistema
String os = "Linux"; // sistema operacional
String vmm = "Xen"; // monitor de maquina virtual
double time_zone = 10.0; // o fuso horario em que esse recurso se
 ↳localiza
double cost = 3.0; // o custo do processamento
double costPerMemory = 0.05; // o custo do uso da memoria
double costPerStorage = 0.001; // o custo do uso do armazenamento
double costPerBw = 0.0; // o custo de uso da largura de banda

// new DatacenterCharacteristics(arquitetura, sistema operacional,
 ↳monitor de maquina virtual, lista de hosts, fuso horario, custo do
 ↳processamento, custo do uso da memoria, custo do uso do
 ↳armazenamento, custo do uso da largura de banda)
DatacenterCharacteristics characteristics = new
 ↳DatacenterCharacteristics(
        arch,
        os,
        vmm,
        hostList,
        time_zone,
        cost,
        costPerMemory,
        costPerStorage,
        costPerBw);
```

O que se fez na Listagem 3 foi definir algumas características, como a arquitetura (**arch**), sistema operacional (**os**), monitor de máquina virtual (**vmm**), fuso horário (**time_zone**), custo do processamento (**cost**), custo do uso da memória (**costPerMemory**), custo

do uso do armazenamento (**costPerStorage**) e o custo do uso da largura de banda (**costPerBw**). A partir das características criadas, pode-se instanciar o *datacenter*, que será criado antes da configuração dos *switches*. A criação do *datacenter* é observada na listagem 4 e possui um nome (**name**), as características definidas anteriormente (**characteristics**), política de alocação de VM (**NetworkVmAllocationPolicy**), classe responsável por alocar uma *VM* ao *host* com menos *Pe* em uso, lista de dispositivos de armazenamento (**storageList**) e o intervalo de escalonamento.

Listagem 4: Criação do *Datacenter*

```
1  LinkedList<Storage> storageList = new LinkedList<Storage>(); // lista
    ↳ de storages, ainda vazia
2
3  NetworkDatacenter datacenter = null;
4  try {
5      // new NetworkDatacenter(nome, caracteristicas, new
    ↳NetworkVmAllocationPolicy(lista de hosts), lista de storages,
    ↳intervalo de escalonamento)
6      datacenter = new NetworkDatacenter(
7              name,
8              characteristics,
9              new NetworkVmAllocationPolicy(hostList),
10             storageList,
11             0);
12
13 } catch (Exception e) {
14     e.printStackTrace();
15 }
```

Em posse do *datacenter*, é criada a rede. O NetworkCloudSim disponibiliza três classes diferentes para *switches*. A primeira é a *RootSwitch*, que conecta o *datacenter* a uma rede externa e direciona os pacotes externos aos *switches* do *datacenter*. O segundo tipo de *switch* é o *EdgeSwitch*, que se conecta diretamente aos *hosts* e direciona os pacotes a outros *switches*. Por último, pode-se criar um *AggregateSwitch*, que direciona os pacotes entre os *switches*. Todos os *switches* necessitam dos dados de nome, nível e o *datacenter*, criado anteriormente. A criação de uma rede com um *switch* de cada tipo é observada na listagem 5.

Listagem 5: Criação dos *Switches*

```
1  // new RootSwitch(nome, nivel, datacenter)
```

```
2  RootSwitch rootSwitch = new RootSwitch("Root0", NetworkConstants.
    ↳ROOT_LEVEL, datacenter);
3
4  // new AggregateSwitch(nome, nivel, datacenter)
5  AggregateSwitch aggregateSwitch = new AggregateSwitch("Aggregate1",
    ↳NetworkConstants.Agg_LEVEL, datacenter);
6
7  EdgeSwitch edgeSwitches[] = new EdgeSwitch[2];
8
9  for(int i = 0; i < 2; i++) {
10     // new EdgeSwitch(nome, nivel, datacenter)
11     edgeSwitches[i] = new EdgeSwitch("Edge" + (i + 2),
    ↳NetworkConstants.EDGE_LEVEL, datacenter);
12
13     // adiciona um switch no sentido Edge -> Aggregate
14     edgeSwitches[i].uplinkswitches.add(aggregateSwitch);
15
16     // adiciona um switch no sentido Aggregate -> Edge
17     aggregateSwitch.downlinkswitches.add(edgeSwitches[i]);
18
19     // adiciona o novo switch ao datacenter
20     datacenter.Switchlist.put(edgeSwitches[i].getId(), edgeSwitches[i
    ↳]);
21 }
22
23 // adiciona um switch no sentido Root -> Aggregate
24 rootSwitch.downlinkswitches.add(aggregateSwitch);
25
26 // adiciona um switch no sentido Aggregate -> Root
27 aggregateSwitch.uplinkswitches.add(rootSwitch);
28
29 /* Cria as ligacoes necessarias para que os pacotes encontrem os
    ↳switches,
30  * datacenters e hosts */
31 for (Host host : datacenter.getHostList()) {
32     NetworkHost networkHost = (NetworkHost) host;
33     networkHost.bandwidth = NetworkConstants.BandWidthEdgeHost;
34     int switchIndex = host.getId();
35
36     // conecta um host a um switch
37     edgeswitch[switchIndex].hostlist.put(host.getId(), networkHost);
38
39     // diz ao datacenter a conexao entre um host e um switch
```

```
40      datacenter.HostToSwitchid.put(host.getId(), edgeswitch[
    ↳switchIndex].getId());
41
42      // conecta o switch ao host
43      networkHost.sw = edgeswitch[switchIndex];
44  }
```

Com toda a estrutura criada, pode-se criar o *broker*, que age como um intermediário entre o usuário e a estrutura de rede e máquinas criadas, por isso é definido junto a uma referência à lista de VMs e o *datacenter*. A listagem 6 ilustra a criação do *broker*.

Listagem 6: Criação do Broker

```
1  // broker e membro da classe
2  broker = new MyDatacenterBroker("Broker");
3
4  // vmList e membro da classe
5  vmList = new ArrayList<NetworkVm>();
6
7  broker.setLinkDC(datacenter);
8  broker.submitVmList(vmList);
```

Por último, dentro da classe *MyDatacenterBroker*, são criadas as máquinas virtuais, concluindo a definição e criação do ambiente de nuvem. A criação das máquinas virtuais é demonstrada na listagem 7 e cada VM necessita de um identificador da VM (**vmID**), MIPS (**mips**), tamanho da imagem (**size**), memória utilizada pela VM (**ram**), largura de banda utilizada (**bw**), quantidade de *Pes* usados por cada VM (**qtdePe**) e o monitor de máquina virtual (**vmm**).

Listagem 7: Criação das Máquinas Virtuais

```
1  private void CreateVMs(int datacenterId) {
2      // serao criadas duas vezes mais VMs que a quantidade de hosts
3      int qtdeVM = linkDC.getHostList().size() * NetworkConstants.
    ↳maxhostVM;
4      for (int i = 0; i < qtdeVM; i++) {
5          int vmID = i;
6          int mips = 10;
7          // tamanho da imagem da VM
8          long size = 10000;
9          // memoria utilizada
10         int ram = 512;
```

```
11          // largura de banda utilizada
12          long bw = 1000;
13          // quantidade de PEs usados por cada maquina virtual
14          int qtdePe = 2;
15          // nome do monitor de maquina virtual
16          String vmm = "Xen";
17
18          // new NetworkVm(id, id do broker, mips, quantidade de PEs,
    ↳memoria, largura de banda, tamanho da imagem, nome do vmm,
    ↳escalonador de tarefas)
19          NetworkVm vm = new NetworkVm(
20                  vmID,
21                  getId(),
22                  mips,
23                  qtdePe,
24                  ram,
25                  bw,
26                  size,
27                  vmm,
28                  new NetworkCloudletSpaceSharedScheduler());
29
30          // cria a VM pelo datacenter
31          linkDC.processVmCreateNetwork(vm);
32          getVmList().add(vm);
33          // associa uma VM a um datacenter
34          getVmsToDatacentersMap().put(vmID, datacenterId);
35          getVmsCreatedList().add(VmList.getById(getVmList(), vmID));
36      }
37  }
```

A.1 Criação das tarefas (Cloudlets)

Anteriormente, foi descrita e criada a estrutura do *datacenter*. Para executar um *workflow*, que é a estrutura que simula o envio e recebimento de mensagens entre VMs, agora é preciso descrever as tarefas (**Cloudlets**) e os estágios das tarefas (**TaskStage**), que enviarão mensagens entre si. O diagrama de classes dessa etapa é mostrado na Figura 1.

A classe *MyAppCloudlet* é responsável pela criação dos *Cloudlets* e *TaskStage*. Essa classe é instanciada em *MyDatacenterBroker*, no método *createVmsInDatacenterBase*. A

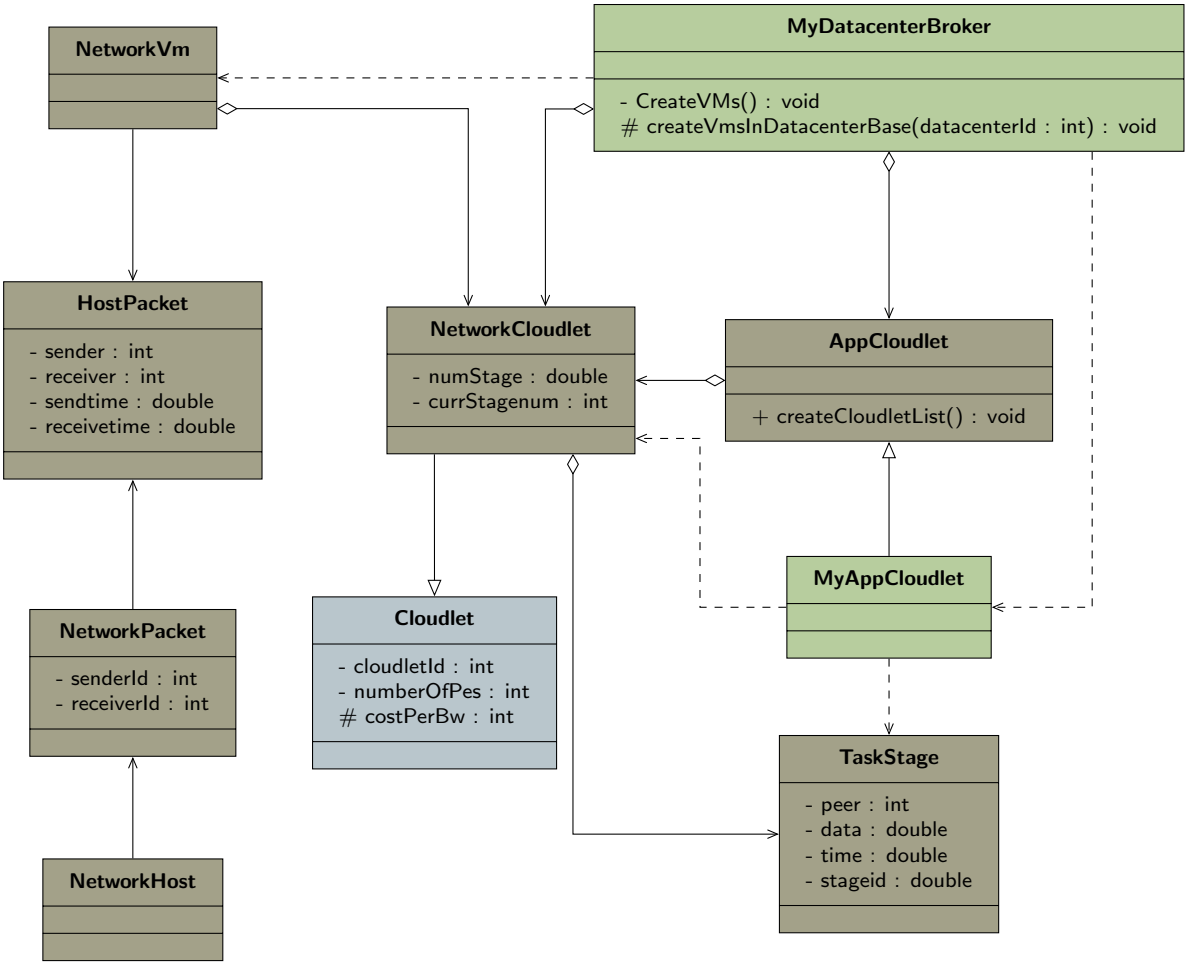


Figura 1: Diagrama parcial do NetworkCloudSim - Criação de cloudlets

criação da classe *MyAppCloudlet* é mostrada na listagem 8.

Listagem 8: Criação do AppCloudlet

```
1     // new MyAppCloudlet(tipo do app, id do app, deadline, numero de
    ↳VMs, id do broker)
2     this.getAppCloudletList().add(new MyAppCloudlet(AppCloudlet.
    ↳APP_Workflow, NetworkConstants.currentAppId++, 0, 0, getId()));
```

Quando o *AppCloudlet* é criado, seu construtor deve iniciar o número de VMs (**numbervm**), como na listagem 9.

Listagem 9: Construtor do AppCloudlet

```
1 public MyAppCloudlet(int type, int appID, double deadline, int
  ↳numbervm, int userId) {
2     super(type, appID, deadline, numbervm,userId);
3
```

```
4       this.numbervm = 4;
5   }
```

Nesse ponto, já se pode criar os *NetworkCloudlets*, que são as tarefas, e os *TaskStage*, que são os estágios das tarefas. A criação é mostrada na listagem 10, em que um número definido de *Cloudlets* é instanciado, cada um atribuído a uma VM. O objeto **NetworkCloudlet** é criado com os dados de identificação, tempo de execução, quantidade de *Pes* utilizados, tamanho antes e depois da execução, tamanho da memória, modelo de utilização da CPU, modelo de utilização da memória RAM e o modelo de utilização da largura de banda, sendo o modelo de utilização um objeto que define a forma com que os recursos serão utilizados. Depois da criação do *Cloudlet*, segue a criação dos estágios, que necessitam dos dados de tipo do estágio (execução, enviar ou receber), dado (se houver um dado específico a ser enviado), tempo de execução (utilizado para o tipo de estágio de execução), identificador do estágio, memória utilizada, identificação da VM de destino ou origem e identificação do *Cloudlet* de destino ou origem.

Listagem 10: Criação das tarefas e estágios

```
1   @Override
2   public void createCloudletList(List<Integer> vmIdList) {
3       int stageID = 0;
4       for(int i = 0; i < numbervm; i++) {
5           // o modelo de utilizacao dos recursos
6           UtilizationModel utilizationModel = new UtilizationModelFull
    ↳();
7
8           // new NetworkCloudlet(id do cloudlet, milhoes de instrucoes,
    ↳ quantidade de PEs, tamanho de arquivo antes da execucao, tamanho
    ↳de arquivo depois da execucao, tamanho da memoria, modelo de
    ↳utilizacao da CPU, modelo de utilizacao da RAM, modelo de
    ↳utilizacao da largura de banda)
9           NetworkCloudlet cloudlet = new NetworkCloudlet(
    ↳NetworkConstants.currentCloudletId++, 250, 2, 256, 256, 1024,
    ↳utilizationModel, utilizationModel, utilizationModel);
10          cloudlet.setUserId(userId);
11          // momento em que a cloudlet foi submetida
12          cloudlet.submittime=CloudSim.clock();
13          // inicia o numero de estagio
14          cloudlet.currStagenum=-1;
15          cloudlet.setVmId(vmIdList.get(i));
16          /* adiciona um estagio inicial de execucao para todos os
```

```
    ↳cloudlets
17          * cloudlet.stages - lista de estagios do cloudlet
18          * new TaskStage(tipo do estagio, dado, tempo de execucao, id
    ↳ do estagio, memoria utilizada, id da VM, id do cloudlet) */
19          cloudlet.stages.add(new TaskStage(NetworkConstants.EXECUTION,
    ↳ 0, 100, stageID++, memory, vmIdList.get(0), cloudlet.getCloudletId
    ↳()));
20
21          stageID = 0;
22          if(i == 0) {
23              // new TaskStage(tipo do estagio, dado, tempo de execucao
    ↳, id do estagio, memoria utilizada, id da VM de destino, id do
    ↳cloudlet de destino)
24              cloudlet.stages.add(new TaskStage(NetworkConstants.
    ↳WAIT_SEND, 0, 0, stageID++, memory, vmIdList.get(1), 1));
25              // new TaskStage(tipo do estagio, dado, tempo de execucao
    ↳, id do estagio, memoria utilizada, id da VM do remetente, id do
    ↳cloudlet do remetente)
26              cloudlet.stages.add(new TaskStage(NetworkConstants.
    ↳WAIT_RECV, 0, 0, stageID++, memory, vmIdList.get(3), 3));
27          } else if(i == 1) {
28              cloudlet.stages.add(new TaskStage(NetworkConstants.
    ↳WAIT_SEND, 0, 0, stageID++, memory, vmIdList.get(2), 2));
29              cloudlet.stages.add(new TaskStage(NetworkConstants.
    ↳WAIT_RECV, 0, 0, stageID++, memory, vmIdList.get(0), 0));
30          } else if(i == 2) {
31              cloudlet.stages.add(new TaskStage(NetworkConstants.
    ↳WAIT_SEND, 0, 0, stageID++, memory, vmIdList.get(3), 3));
32              cloudlet.stages.add(new TaskStage(NetworkConstants.
    ↳WAIT_RECV, 0, 0, stageID++, memory, vmIdList.get(1), 1));
33          } else if(i == 3) {
34              cloudlet.stages.add(new TaskStage(NetworkConstants.
    ↳WAIT_SEND, 0, 0, stageID++, memory, vmIdList.get(0), 0));
35              cloudlet.stages.add(new TaskStage(NetworkConstants.
    ↳WAIT_RECV, 0, 0, stageID++, memory, vmIdList.get(2), 2));
36          }
37
38          // adiciona o cloudlet criado
39          clist.add(cloudlet);
40      }
41  }
```

A execução desse código, resultará em 4 tarefas, trocando mensagens entre si, com

a tarefa 0 enviando uma mensagem para o *Cloudlet* 1 e imediatamente aguardando uma mensagem da tarefa 3, enquanto que o *Cloudlet* 1 envia uma mensagem para a tarefa 2 e recebe a mensagem do *Cloudlet* 0. A tarefa 2 envia uma mensagem para 3 e recebe a mensagem de 1. Finalmente, o *Cloudlet* 3 envia a mensagem para 0 e recebe a mensagem de 2.

Com essa demonstração básica, é possível modelar serviços de nuvem com a possibilidade de observar o funcionamento do serviço ao se utilizar programas que enviam e recebem mensagens, e com o estágio do tipo *EXECUTION*, também pode-se adicionar um tempo de execução para cada tarefa.